Declarative Performance Testing Automation

Automating Performance Testing for the DevOps Era

Doctoral Dissertation submitted to the
Faculty of Informatics of the Università della Svizzera Italiana
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

presented by
Vincenzo Ferme

under the supervision of
Prof. Cesare Pautasso

February 2021

Dissertation Committee

Prof. Walter Binder	Università della Svizzera italiana
Prof. Mauro Pezzè	Università della Svizzera italiana
Prof. Lionel Briand	University of Luxembourg & University of Ottawa
Prof. Dr. Dr. h. c. Frank Leymann	University of Stuttgart

Dissertation accepted on 7 February 2021

Research Advisor	PhD Program Director
Prof. Cesare Pautasso	**Prof. Dr. Walter Binder, Prof. Dr. Silvia Santini**

I certify that except where due acknowledgment has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Vincenzo Ferme
Lugano, 7 February 2021

Dedicated to people who backed me up throughout this long journey, and to me for making it to the end.

The ultimate reason for setting goals is to entice you to become the person it takes to achieve them.

Jim Rohn

Abstract

Recent trends in industry show increasing adoption of Development and Operations (DevOps) practices. Reasons for increasing DevOps adoption are the focus on the creation of cross-functional teams, and the ability to release high-quality software at a fast pace. Alongside the adoption of DevOps, performance testing continues to evolve to meet the growing demands of the modern enterprise and its need for automation. As DevOps adoption continues and self-service environment provisioning becomes commonplace in Information Technology (IT) departments, more developers will be working on executing performance tests, to ensure the quality of released services satisfies users' expectations while constraining the resources needed to do so.

Modeling and automated execution of performance tests are time-consuming and difficult activities, requiring expert knowledge, complex infrastructure, and a rigorous process to guarantee the quality of collected performance data and the obtained results. Currently available performance testing approaches are not well integrated with DevOps practices and tools and are often focusing only on specific needs of performance testing modeling and automation.

A recent survey by the Standard Performance Evaluation Corporation (SPEC) Research Group (RG) on DevOps reported the need for a new paradigm for performance activities to be successfully integrated with DevOps practices and tools, such as the one proposed by Declarative Performance Engineering (DPE). Previous studies reported successful applications of DPE to DevOps contexts, due to the opportunity to model the performance testing domain knowledge as a first-class citizen and its ability to offer different levels of abstraction to different people relying on it.

In this dissertation, we introduce a "*Declarative Approach for Performance Tests Execution Automation*" enabling the continuous and automated execution of performance tests alongside the Continuous Software Development Lifecycle (CSDL), an integral part of DevOps practices. We contribute an automation-oriented catalog of performance test types and goals and a description of how they fit in different moments of the CSDL, a declarative Domain

Specific Language (DSL) enabling the declarative specification of performance tests and their automated orchestration processes alongside the CSDL, and a framework for end-to-end automated performance testing of RESTful (RESTful) Web services and Business Process Model and Notation 2.0 (BPMN 2.0) Workflow Management Systems (WfMSs) relying on the contributed DSL. We evaluate the proposed DSL by conducting an expert review targeting its overall expressiveness and suitability for the target users, perceived usability and effort, and reusability of specified tests. We also perform a summative evaluation of the DSL's usability in terms of learnability, and reusability of test specifications. The surveys confirm the proposed approach is valid for the aims it has been built for, and it is considered on average good for all the evaluated usability dimensions. We evaluate the implemented framework by performing iterative reviews of the different versions of the framework, and a comparative evaluation of the proposed framework's features compared to state-of-the-art available solutions. The iterative reviews led to many improvements due to the received constructive feedback, while the comparative evaluation showed no other solutions similar to the proposed one are available in the literature. We assess the overall contributed solution by executing a large number of case studies, by collaborating with other researchers in extending both the DSL and the framework.

Acknowledgments

Almost four years as a PhD candidate at the Architecture, Design and Web Information Systems Engineering Group (Software Institute, Università della Svizzera italiana (USI)), six months as visiting researcher at the Software Quality and Architecture Group (University of Stuttgart) and more than two years as Cloud Native Tech Lead in Kiratech, working with enterprises in Switzerland and Italy. During this time I developed my dissertation collaborating with many people, experiencing failures and successes, and facing multiple challenges in different contexts.

The first person who made my PhD journey possible is my advisor, Cesare Pautasso. We met the first time in late 2013 for the interviews to get accepted for a PhD position, after I applied for it and had the initial interview with Michele Lanza. I will never thank you enough for advising me throughout these years, for helping me sort out my (sometimes too many and fuzzy) ideas, and for working together to improve and execute them iteratively. I am also thankful for making me realize by experience, all the work there is behind every single lecture.

A warm thank you to my colleagues Masiar Babazadeh, Andrea Gallidabino, Ana Ivanchikj, and Vasileios Triglianos for accepting my idiosyncrasies, and sharing with me most of my PhD journey, some travels around the World, and for joining me at my Wedding party. A particular thank you to Ana Ivanchikj for working together on many research work.

A big thank you also to my collegues Steffen Becker, Thomas F. Düllmann, Markus Frank, Floriment Klinaku, Dušan Okanović, Teerat Pitakrat, André van Hoorn, and Henning Schulz for welcoming me in Stuttgart and sharing with me great professional and personal moments.

A grateful thank you to Oliver Kopp, Marigianna Skouradaki, and Frank Leymann, and other colleagues from the Institute of Architecture of Application Systems at the University of Stuttgart, and Matthias Geiger, Simon Harrer, and Jörg Lenhard from the University of Bamberg. We attended great research group retreats in Lauterbad and shared authentic German culture

experiences and great bonding time. A particular thank you to Marigianna Skouradaki for working together on many research work as part of the BenchFlow project.

Thanks also to Janine Caggiano, Elisa Larghi, Danijela Milicevic, Paolo Schmidt, Giacomo Toffetti-Carughi, and Jacinta Vigini for being always responsive and supportive throughout my journey at Universitá della Svizzera italiana.

A wholeheartedly thank you to my entire extended family for always being supportive of my life and work choices, and for being present in happy times and reassuring in difficult times. Some of you did not live long enough to see the start and/or the end of this PhD journey in person, but I am sure you are with me and you are assisting me with your love.

A friendly thank you also to my friends for sharing my life journey. Some of you have been always there as we grew up together, some of you shared with me the high-school experience, my Bachelor studies in Naples, my Master studies in Milan, my PhD studies and experiences in different parts of the World, and my work experience while living in Como. No matter when our friendship started you made me feel at home, and you have always been with me and you are still here. Your unconditional support proved monumental towards the success of my journey and I am sure it will be for the continuation of the same.

I wish to show my gratitude also to all the teachers, assistants, and classmates I encountered during my academic journey. I am what I am also because of you, and without the support of some of you, I would have not started, and definitely, I would have not reached the end of this great PhD experience.

Thank you to the students who decided to share their thesis experience with me: Marco Argenti, Francesco Berla, Gabriele Cerfoglio, Simone D'Avico, Jesper Findahl, Ana Ivanchikj, Marco Nivini, Manuel Palenga, and Christoph Zorn. I learned something from each one of you, and I hope you all learned something useful from me. Thank you also to all the students I taught during my experience as a teaching assistant. I learned a lot from dealing with so many different people and cultures.

Thank you also to all the other people I met during my academic journey, most of whom while traveling at research retreats and many conferences around the World. Especially, collaborating with researchers in the SPEC RG DevOps Performance Working Group has been a great source of inspiration. Our joint work broadened my view on performance engineering. Thank you to my colleagues in Kiratech and the great people I met while working in Kiratech as well. I am learning a lot from you. Thanks also to all the participants

who replied to the surveys I shared for getting feedback on my work. All your inputs helped me shape this dissertation.

I also want to show my gratitude to the members of my dissertation committee: Prof. Walter Binder, Prof. Lionel Briand, Prof. Dr. Dr. h. c. Frank Leymann, and Prof. Mauro Pezzè. Thank you for the time you invested in providing insightful feedback on my dissertation.

Last but—needless to say—not least, there are not enough words to thank my wife Caterina for staying close to me during my entire academic journey, from my Bachelor to my PhD. In the last years, I struggled to bear with myself. I can not imagine how difficult it must have been for you. Thank you for always being supportive and enduring my absences.

Contents

Figures

Tables

Listings

Part I

Prologue

Chapter 1

Introduction

In this chapter, we present the context in which this PhD is developed and the motivations of our work. Based on these we express our problem statement and research goals and then discuss our contribution to the current state-of-the-art in the area, as well as how we evaluate our work. We conclude the chapter by offering a throughout thesis outline and an exhaustive list of publications we contribute as part of this work and as part of the overall career.

1.1 Context and Motivation

Continuous Software Development Lifecycle (CSDL) is very common nowadays [Fitzgerald and Stol, 2017]. Companies are adopting agile practices, such as Development and Operations (DevOps) [Jabbari et al., 2016], in their software development processes, to introduce more and more automation and velocity in their software development and release lifecycles, characterized by continuous improvement based on feedback [Humble and Farley, 2010; Itkonen et al., 2017]. Agile practices, such as DevOps, are characterized by:

a) continuous development and release of applications following a Continuous Software Development Lifecycle, often organized as multiple services communicating with each other and contributing cohesive functionalities to the overall application;

b) pervasive automation as a medium to enable continuous development and release;

c) the presence of many small, self-organized, and cross-functional teams depending on each other;

d) service ownership and responsibility from development to production assigned to those teams;

e) professional figures having diversified roles and heterogeneous knowledge [Wettinger et al., 2016].

In agile, fast, and continuous development lifecycles, software performance testing is fundamental to confidently release continuously improved software versions [Itkonen et al., 2017; Bosch, 2014], and more and more Information Technology (IT) practitioners are approaching performance testing [PractiTest, 2020; Tricentis, 2020]. It also introduces different challenges [Brunnert et al., 2015; Walter et al., 2016]. Software performance testing needs to ensure that characteristics of the System Under Test (SUT) related to speed (e.g., responsiveness, and user experience), scalability (e.g., capacity, load, and volume), and stability (e.g., consistency, and reliability) reach, maintain and adjust to the defined and evolving performance and acceptance criteria. Although the modern approach to software development makes performance testing more difficult [Eismann et al., 2020], and available solutions tend to skip in-depth performance testing [Dynatrace, 2019], we consider performance testing in this context very important, especially for systems having clients that expect agreed Service Level Agreement (SLA) to be respected, for which software providers have to demonstrate that the system can hold the SLAs. Performance testing requires different skill-sets which comprise: "writing test scripts, monitoring and analyzing test results, tweaking the application and repeating the whole process again" [Murty, 2016]. As argued by us [Ferme and Pautasso, 2017], and by other researchers (e.g., [Brunnert et al., 2015]), execution of performance tests should be automated, flexible, context- and business-aware, so that it can cope with the velocity introduced by modern life-cycles and contribute to the validation of the released software quality. As a recent survey [Bezemer et al., 2019] by the Standard Performance Evaluation Corporation Research Group on DevOps reported, the complexity of performance engineering approaches is seen as a barrier for wide-spread adoption by practitioners [Streitz et al., 2018], because performance engineering approaches are not lightweight enough and are not well integrated with existing tools in the DevOps ecosystem. The survey also reported a new paradigm is needed for performance activities to be successfully integrated into DevOps practices and tools, such as the one proposed by Declarative Performance Engineering (DPE). Two of the main reported advantages of approaches such as DPE are the opportunity to model the performance testing domain knowledge as a first-class citizen and its ability to offer different levels of abstraction to different people relying on it.

Two outstanding research activities proposing DPE as an approach to cope with the mentioned challenges, are the DECLARE project[1], and the ContinuITy project[2]. The DECLARE project focuses mainly on enabling the possibility of declaratively querying performance knowledge that has been collected and modeled by other systems. The ContinuITy project focuses on dealing with the challenges of continuously updating performance tests, by leveraging performance knowledge of services collected and modeled from production environments. These tests can be then continuously executed during the software development lifecycle by other frameworks, to validate the performance of the system over time and potentially identifying significant performance issues in the developed software and/or its third-party dependencies, early in the project. A third relevant research activity in the context of this thesis is the TESTOMAT project[3], proposing to "advance the state-of-the-art in test automation for software teams moving towards a more agile development process".

The above-mentioned projects define the context in which this PhD work is carried on, and we fill an identified research gap about the need of codifying performance tests knowledge, abstracting away complexity, and offering it as a service to the different people and systems needing access to it, by the means of automation.

1.2 Problem Statement and Research Goals

In this thesis, we focus on the performance test specification and execution facet of performance engineering and we contribute to enabling DPE by looking at the automation of performance testing activities. The formulation of the problem statement is as follows:

[1] http://www.dfg-spp1593.de/declare, last visited on February 7, 2021
[2] https://continuity-project.github.io, last visited on February 7, 2021
[3] https://itea3.org/project/testomatproject.html, last visited on February 7, 2021

To design new methods and techniques for the declarative specification of performance tests and their automation processes, and to provide models and frameworks enabling continuous and automated execution of performance tests, in particular for the case of RESTful (RESTful) Web services and Business Process Model and Notation 2.0 (BPMN 2.0) Workflow Management Systems (WfMSs) deployed as Docker services. The proposed methods, techniques, models, and frameworks should be i) suitable to be used alongside the CSDL, and ii) open for extension and integration in the overall Declarative Performance Engineering vision.

The main target users of the proposed methods and techniques are practitioners involved in the CSDL, and in particular developers, software testers, quality assurance engineers, DevOps engineer, and operations engineers. According to recent surveys about software testing [PractiTest, 2020; Tricentis, 2020], the mentioned practitioners are more and more involved in defining and executing functional and non-functional tests, including performance tests.

1.2.1 Research Goals

We set four main research goals to accomplish our objectives stated above. From the first goal about defining an automation-oriented catalog of performance tests to the last one about models, languages, and tools to execute performance tests in CSDL, we develop new methods, techniques, and tools enabling declarative performance engineering in the context of automated execution of performance tests.

R.G. 1 - Automation-oriented Performance Tests Catalog

The first research goal (R.G. 1) is about defining an automation-oriented performance test catalog presenting the different kinds of performance tests that get executed against software systems. We take into consideration the factors influencing the selection and the complexity of the executed tests from the point of view of their automation, and the contexts in which they are executed. Performance tests are of many different types and are influenced by many different factors such as the system's architecture and its deployment environment. Additionally, the development process in which they have to be executed has an impact on the types of performance tests that get defined. We identified the need of collecting and rationalize the knowledge on performance test types, by providing a catalog looking at factors influencing the definition

and automation of performance tests from different points of view, such as the system architecture, technologies, deployment environment, and software development lifecycle. By defining a clear catalog, we enable the opportunity to define and model the data and the processes needed to enable a declarative specification of performance tests and automated execution of the same.

R.G. 2 - Declarative Specification Language

The second research goal (R.G. 2) is about designing and implementing a Domain Specific Language (DSL) enabling declarative specification of performance test automation processes. The DSL we propose allows users to declaratively specify performance tests and their automation processes collecting the users' intent and offering an abstraction layer for the users to specify all the data needed for the automated execution of performance tests out of the proposed catalog. This is our first contribution towards defining a DPE approach for performance testing, and it is an important one because of the need to embrace people with diversified skill sets by enabling them to rely on and codify their performance testing needs using an expressive DSL.

R.G. 3 - Automated Execution of Declarative Performance Tests

The third research goal (R.G. 3) is about providing a framework for automated execution of declarative performance testing. The framework codifies the models behind the provided DSL and can be programmed using such DSL to execute performance tests and collect performance data following defined and validated automation processes. The framework enables the scheduling of performance tests that are managed by an expert system. The expert system can be configured using the DSL to adjust the expected behavior of the automation process to the user's and test's needs. Providing a framework is a needed step towards enabling DPE in the context of the thesis, because for DPE to be successful the complexity of executing automated performance tests has to be taken away from users' as much as possible and offered to them as a service [Hilton et al., 2016]. Recent studies about DevOps practices show that tools are widely used in the process, and developers and software operations engineers heavily rely on tools automating and assisting their work [Logz.io, 2016; Goparaju et al., 2012].

R.G. 4 - Executing Performance Testing in CSDL

The fourth and last research goal (R.G. 4) is about enabling the integration of the proposed DSL for automated performance test execution, in the CSDL. Although the DSL and the framework already provide users' with a DPE platform for automated performance test execution, in modern software development lifecycles it is required that such solutions are well integrated with CSDL processes and tools driving the automated software delivery process. We enable the integration of the proposed DPE solution with the CSDL, by providing additional declarative abstractions in the DSL allowing the user to specify test suites, an analysis of a proposed mapping of CSDL events to selected performance test suites supported by the contributed DSL, and a Command-line Interface (CLI) utility to trigger different declaratively specified test suites in different moments of the CSDL. The triggered test suites then provide back to the CSDL with data that can be relied on to define CSDL-level quality gates.

Overall the proposed research goals contribute to the first DPE approach for continuous and automated execution of performance tests alongside the CSDL, and embrace DevOps goals by enabling the end-to-end execution of performance tests, including SUT lifecycle management.

1.3 Contributions and Evaluation Summary

We contribute models, analysis, tools, and artifacts to answer the stated research goals.

In this dissertation we mainly focus on performance testing of RESTful Web services and BPMN 2.0 WfMSs deployed using a deployment descriptor relying on Docker containers [Docker Inc., 2013], thus our contributions refer mainly to these kinds of SUTs. RESTful Web services and BPMN 2.0 WfMSs are systems usually experiencing many customers interacting with them, both humans and other systems. For these systems is important to ensure performance is guaranteed. The choice in referring to systems deployed using Docker is motivated by the fact that Docker adoption was quickly rising[4] at the time we started our work, implying that many developers may already have defined Docker files for their software products. Docker and Docker-based deployment infrastructures adoption rose exponentially in the years following our decision [Hackernoon,

[4] https://www.datadoghq.com/docker-adoption/, updated June 2014-2018. Last visited on February 7, 2021

2020], further validating our choice in relying on this technology.

In the following subsections we describe the different contributions, how they are mapped to the research goals presented in Sect. 1.2.1, how we evaluate them, and how they contribute to the overall proposed DPE approach for continuous and automated execution of performance tests alongside the CSDL.

1.3.1 Modeling and Analysis

As part of the work done for the R.G. 1, we contribute an analysis of different factors influencing the definition, selection, and configuration of performance tests. Out of the carried analysis, we propose an automation-oriented performance test catalog taking into consideration the identified factors and presenting an overview of the different kinds of performance tests analyzing their configurations (e.g., input data, and SUT deployment conditions) and execution processes to enable declarative specification and automation of the same.

To enable the declarative specification of tests, we propose a DSL as part of the contribution related to the R.G. 2. The proposed DSL offers declarative abstractions to specify different kinds of performance tests identified in the catalog result of R.G. 1. The DSL's expressiveness enables users to either specify performance tests and configure their automation process, or reuse already defined performance tests provided as templates to reduce the complexity of defining and executing performance tests against the SUT under development. Relying on the proposed DSL, users can state their performance test goals, configure how to deploy and interact with the SUT, define measurements data to collect and metrics to compute, and parameterize the automation process to decide when to terminate the execution and how to assess the quality of the SUT from the performance point of view. For a selected subset of the identified performance tests in the contributed catalog, we provide abstractions in the DSL for the users to declaratively specify these performance tests, as well as provide templates to be directly reused by the users. We evaluate the proposed DSL's usability [Rodrigues et al., 2018] in terms of its learnability, usability, and reusability by performing a summative evaluation, also by comparing it with standard imperative approaches, like the one proposed by JMeter [JMeter, 1998]. We also perform a survey-based expert review, to assess the overall DSL model, its expressiveness, perceived usability and effort, as well as the overall suitability of the approach for the target users, also when compared to standard imperative approaches. The expressiveness of the DSL is also evaluated in multiple case studies in the performance testing domain, and by iteratively

enhancing it based on continuous feedback collected in collaborating with other researchers, as well as during different Master and Bachelor theses.

As part of the work done to answer the R.G. 4, we extend the DSL as well as provide additional abstractions and models to enable continuous execution of performance tests alongside the CSDL. The proposed abstractions allow users to map CSDL events to different performance test suites of selected performance tests, that are triggered in different moments of the CSDL. The triggered test suites provide back to the CSDL with data that can be relied on to define CSDL-level quality gates. We also contribute an analysis of a proposed mapping of CSDL events to selected performance test suites supported by the contributed DSL. We evaluate the proposed CSDL extension to the DSL as part of the survey-based expert review and in the summative evaluation performed for the DSL.

1.3.2 Tools and Artifacts

As part of the work performed on R.G. 2, we contribute an implementation of the proposed DSL [Ferme and Pautasso, 2018]. The DSL is implemented as a software library meant to be integrated with tools needing to generate performance tests, and tools implemented to read declarative performance test specifications. The language we utilize to present the DSL to users is YAML [Evans, 2001], a de-facto standard for declarative DSLs implementations and tools to be used within CSDL. The DSL implementation is made available as an open-source project on GitHub[5] named BenchFlow. We continuously evaluate and improve the DSL by collaborating with different researchers as part of the SPEC RG on DevOps, performing many case studies utilizing the DSL, as well as by developing integration and abstractions based on the DSL as part of the research work done in projects such as the DECLARE project and the ContinuITy project.

The main tool contribution is the result of the R.G. 3. We design and implement a framework [Ferme and Pautasso, 2016] accepting performance tests specified using the developed DSL, providing the automation needed for enabling the execution of declarative performance tests. The framework is realized by multiple cohesive and interconnected services, collaborating in providing rigorous automation processes to guarantee the quality of collected performance data and the obtained results, automated SUT deployment, scheduling, data collection, and metrics computation facilities, and integration with

[5] https://github.com/benchflow/benchflow, last visited on February 7, 2021

other frameworks we rely on for performance test execution. We implement the different services by relying on state-of-the-art solutions and heavily rely on Docker [Docker Inc., 2013] for both framework's deployment and implementation of some of the offered functionalities, such as automated SUT deployment. The users can access the framework functionalities by relying on a provided CLI, or by directly interacting with exposed Representational State Transfer (REST) Application Programming Interface (API) for accessing the offered functionalities within their CSDL tools. We evaluate the first versions of the framework, by comparing it with state-of-the-art competitive solutions available on the market, especially the open-source one. We then continuously evaluate and improve the framework by utilizing it for many research publications, by deploying it in different research groups across the globe deciding to use our framework for enhancing their ability to carry out their research initiatives that are based on performance tests execution, and by extending and improving it as part of the work done in different successful Master and Bachelor thesis, as well as in different case studies.

To enable the continuous execution of performance tests alongside the CSDL, as part of the R.G. 4 we also extend the CLI to trigger different declaratively specified test suites in different moments of the CSDL, and extend the framework's REST APIs to schedule test suites and the CLI to report back test suites results.

The framework, the CLI and all the related tools and CSDL configurations are made available as an open-source project on GitHub[6]

1.3.3 Other Contributions

Alongside the mentioned main contributions, as part of the research work done during this dissertation we also contribute:

a) an extension of the proposed DSL and the framework in the context of performance-based security testing for the Cloud [Rufino et al., 2020];

b) an extension of the proposed DSL and the framework in the context of generating specifications for elastic services deployed on Kubernetes[7] [Klinaku and Ferme, 2018];

c) a behavioral-driven performance testing definition language enabling human-readable performance tests definition relying on a controlled natural lan-

[6] https://github.com/benchflow/, last visited on February 7, 2021
[7] https://kubernetes.io/, last visited on February 7, 2021

guage [Schulz et al., 2019]. The proposed language is converted to the DSL part of this dissertation for test execution (Sect. 8.3);

d) an approach and a tool for generating performance testing report taking into account the performance test goal stated by the user [Okanović et al., 2019] (Sect. 8.3);

e) in defining a scalability assessment approach for Microservices deployment architectures based on operational profiles and load tests execution [Avritzer et al., 2020] (Sect. 8.4);

f) in enabling the definition and execution of the first performance microbenchmark for BPMN 2.0 WfMSs, as well as establish and enable research on BPMN 2.0 WfMSs' performance analysis, performance comparison and performance regression of such systems (Sect. 8.1).

The above-mentioned contributions stem from the multiple collaborations with international research groups carried on during the PhD, and demonstrate the extensibility of the openness for extension of the proposed approach. Some of those collaborations have been enabled by the initial contributions in defining a DPE approach for performance testing automation, and have been started after we contributed the DPE approach overview at 3rd International Workshop on Quality-aware DevOps (QUDOS'17) [Ferme and Pautasso, 2017], and during the internship based on the Swiss National Science Foundation (SNSF) project "Declarative Continuous Performance Testing for Microservices in DevOps"[8] I have been awarded of.

1.4 Thesis Outline

This thesis is divided into four parts and 11 chapters, plus four appendices.

1.4.1 Part I - Prologue

Part I includes an introduction (Chap. 1) to the topics of this dissertation, an overview of the foundation of performance testing automation (Chap. 2), and an analysis of the state-of-the-art in the fields related to this dissertation (Chap. 3).

[8] http://p3.snf.ch/project-178653, last visited on February 7, 2021

In **Chap. 1** we present the context and motivation for our work, the problem statement, research goals and we provide an overview of the different contributions and evaluations we perform. We conclude the chapter with this thesis outline and an overview of publications we contribute and the overall career up to now.

In **Chap. 2** we discuss the challenges in performance testing automation, and analyze the impact of modern development and release processes, such as DevOps, on the performance testing activities.

In **Chap. 3** we analyse the state-of-the-art in the field relevant to this dissertation. We present what DevOps is and how it is related to performance engineering, other than an overview of the main performance testing techniques for the two main target system of the proposed approach. We then introduce DPE, and analyze the state-of-the-art in the context of performance testing DSLs and performance testing automation framework, with a particular focus on the solutions offering a DPE approach.

1.4.2 Part II - Declarative Performance Testing Automation

Part II presents the core chapters of this dissertation, and includes the automation-oriented performance test catalog (Chap. 4), the declarative DSL enabling the specification of performance test automation processes (Chap. 5), and the BenchFlow framework for declarative performance testing automation (Chap. 6).

In **Chap. 4** we present the automation-oriented performance test catalog we contribute as part of R.G. 1. In doing so we also discuss factors impacting the test definition and the test automation. We also contribute a proposal on how to select different types of performance tests when performance testing is integrated as part of CSDL. The analysis contributes to R.G. 4.

In **Chap. 5** we present the BenchFlow DSL we propose as an answer to R.G. 2 and R.G. 4. The proposed declarative DSL enables the specification of performance test automation process. We present the DSL meta-model, then we introduce the DSL model and in the last part of the chapter we provide examples of how the users can specify declarative performance tests, and test suites, having different goals, as well as implementation details.

In **Chap. 6** we present the BenchFlow framework we contribute as an answer to R.G. 3. The BenchFlow framework enables the automated execution of declarative performance tests. We present the BenchFlow framework design and architecture first and then we provide implementation details.

Being this the core part of the thesis, in this part, we cover all the R.G.s we state in Sect. 1.2.1. In Tab. 1.1 we provide an overview of the mapping between

#	Research Goal	Chapter(s) and Section(s)
1	R.G. 1 - Automation-oriented Performance Tests Catalog	Chapters: Chap. 4, Sections: Sect. 4.3
2	R.G. 2 - Declarative Specification Language	Chapters: Chap. 5, Sections: Sect. 5.4, Sect. 5.6.1, Sect. 5.6.3, Sect. 5.6.4
3	R.G. 3 - Automated Execution of Performance Tests	Chapters: Chap. 6 , Sections: Sect. 6.2.2, Sect. 6.2.4, Sect. 6.2.5, Sect. 6.3.4, Sect. 6.3.5
4	R.G. 4 - Executing Performance Testing in CSDL	Chapters: Chap. 4, Chap. 5, Sections: Sect. 4.4, Sect. 5.5, Sect. 5.6.2, Listing 5.38

Table 1.1. R.G.s Mapped to the Different Chapters and Sections

each R.G. and the main chapters and sections where we discuss contributions to the R.G..

1.4.3 Part III - Evaluation

Part III reports on the different evaluations (Chap. 7) and case studies (Chap. 8) we perform.

In **Chap. 7** we report the results of the expert review (Sect. 7.1) and the summative evaluation (Sect. 7.2) we perform to validate out claims on the proposed declarative DSL and the overall declarative approach for performance testing automation. We then also report the details of the iterative review (Sect. 7.3) we applied while developing the BenchFlow framework and a comparative evaluation (Sect. 7.4) of the BenchFlow framework with other available tools for performance testing automation.

In **Chap. 8** we present relevant case studies in which we applied and ex-

tended the proposed approach while collaborating with other researchers.

1.4.4 Part IV - Epilogue

Part IV analyses the lessons learned (Chap. 9), and summarize the contributions and results obtained in this dissertation (Chap. 10). It then also provides an overview of open challenges and outlook (Chap. 11) in DPE applied to performance testing automation.

In **Chap. 9** presents the lessons learned in developing and extending the proposed approach, as well as related to other people using the proposed approach.

In **Chap. 10** we summarize the contributions and we discuss threats to validity.

In **Chap. 11** we discuss limitations of our approach, and we present the outlook and our vision for DPE approaches for performance testing automation and open challenges for mainstream adoption of the same.

To facilitate access to the information we report in different chapters, each chapter presents a preamble briefly reporting the content discussed in the chapter. For chapters in Part II and III we also provide a *Concluding Remarks* section where we highlight the most relevant information reported in the chapter.

In **Part V** we report relevant artifacts as part of the appendices.

1.5 Publication and Career Overview

The work in this dissertation has been developed building experience over multiple years in academia and industry and collaborating with many great people. I spent almost four years as PhD candidate at the Architecture, Design and Web Information Systems Engineering Group (Software Institute, Università della Svizzera italiana) under the supervision of Cesare Pautasso. I then spent six months as a visiting researcher at the Software Quality and Architecture Group (University of Stuttgart) and more than two years as Cloud Native Tech Lead consultant in Kiratech, working with big enterprises in Switzerland and Italy. During my experience as PhD candidate and visiting researcher, I worked in the research fields of “Workflow Management Systems Performance Benchmarking” and “Performance Testing Automation (Declarative and not)”. During my research career, including the one as a Master student prior to the PhD in the field of “Reverse Engineering and Software Quality”, I had the

opportunity to collaborate with many (more than 40 to February 7, 2021[9]) researchers around the World, on different topics, and publish a fair amount (38 contributions[10]) of research work over the years 2013-2020 at major international conferences and workshops. To February 7, 2021, the research work I contributed has been cited 730 times, with an h-index of 15 and a i10-index of 19[11]. During my recent experience in the industry, I mainly worked as "Cloud Computing and DevOps Advocate" contributing to the digital transformation journeys of big enterprises, providing advanced webinars, training, and working with many of the technologies and processes I rely on for this dissertation.

In the following we report the main contributions in the research areas I worked on during my PhD-related research career, namely "Workflow Management Systems Performance Benchmarking" and "Performance Testing Automation (Declarative and not)". We present the contributions over the years and by classifying them into the following categories: 1) Peer-reviewed journal articles; 2) Peer-reviewed book chapters; 3) Peer-reviewed proceedings, classified in conference proceedings, workshop proceedings, demos, and posters; 4) Research projects; 5) Outreach activities, classified in invited speaker, tutorial, and webinars; 6) Other artifacts with documented use.

	Publications
2019	**Peer-reviewed book chapters** **V. Ferme**, A. Ivanchikj, and C. Pautasso. **IT-Centric Process Automation: Study About the Performance of BPMN 2.0 Engines**. *Empirical Studies on the Development of Executable Business Processes*, pages 167-197, 2019, Springer. [Ferme et al., 2019]

(To be continued)

[9]Source: https://scholar.google.com/citations?user=ju3e7M0AAAAJ&hl=en, last visited on February 7, 2021

[10]Source: https://scholar.google.com/citations?user=ju3e7M0AAAAJ&hl=en, last visited on February 7, 2021

[11]Source: https://scholar.google.com/citations?user=ju3e7M0AAAAJ&hl=en, last visited on February 7, 2021

	Publications
2018	**Peer-reviewed conference proceedings - Conference proceedings** G. Rosinosky, C. Labba, **V. Ferme**, S. Youcef, F. Charoy, and C. Pautasso. **Evaluating Multi-tenant Live Migrations Effects on Performance**. *In Proc. of the 16th Confederated International Conferences "On the Move to Meaningful Internet Systems" (OTM '18)*, pages 61-77, 2018, Springer. [Rosinosky et al., 2018]
2017	**Peer-reviewed conference proceedings - Conference proceedings** **V. Ferme**, M. Skouradaki, A. Ivanchikj, C. Pautasso, and F. Leymann. **Performance Comparison Between BPMN 2.0 Workflow Management Systems Versions**. *In Proc. of the 18th International Conference on Enterprise, Business-Process and Information Systems Modeling (BPMDS '17)*, pages 1-15, 2017, Springer. [Ferme et al., 2017b] - Presentation: https://www.slideshare.net/aivancik/bpmds17-performance-comparison-between-bpmn-20-wfms-versions, last visited on February 7, 2021 A. Ivanchikj, **V. Ferme**, and C. Pautasso. **On the Performance Overhead of BPMN Modeling Practices**. *In Proc. of the 15th International Conference on Business Process Management (BPM '17)*, pages 216-232, 2017, Springer. [Ivanchikj et al., 2017] - Presentation: https://www.slideshare.net/aivancik/on-the-performance-overhead-of-bpmn-modeling-practices, last visited on February 7, 2021 S. Harrer, J. Lenhard, O. Kopp, **V. Ferme**, and C. Pautasso. **A Pattern Language for Workflow Engine Conformance and Performance Benchmarking**. *In Proc. of the 22nd European Conference on Pattern Languages of Programs (EuroPLoP '17)*, pages 1-46, 2017, ACM. [Harrer et al., 2017] **Peer-reviewed conference proceedings - Workshop proceedings** J. Lenhard, **V. Ferme**, S. Harrer, M. Geiger, and C. Pautasso. **Lessons Learned from Evaluating Workflow Management Systems**. *In Proc. of the 13th International Workshop on Engineering Service-Oriented Applications and Cloud Services (WESOACS '17)*, pages 215-227, 2017, Springer. [Lenhard et al., 2017]

(To be continued)

Publications

Contributions to international conferences - Posters

V. Ferme, J. Lenhard, S. Harrer, M. Geiger, and C. Pautasso. **Workflow Management Systems Benchmarking: Unfulfilled Expectations and Lessons Learned**. *In Proc. of the 39th International Conference on Software Engineering (ICSE '17)*, pages 379-381, 2017, IEEE. [Ferme et al., 2017a] - Poster: http://design.inf.usi.ch/sites/default/files/biblio/benchflow-icse2017-poster.pdf, last visited on February 7, 2021

Peer-reviewed conference proceedings - Conference proceedings

2016

V. Ferme, A. Ivanchikj, C. Pautasso, M. Skouradaki, and F. Leymann. **A Container-centric Methodology for Benchmarking Workflow Management Systems**. *In Proc. of the 6th International Conference on Cloud Computing and Service Science (CLOSER 2016)*, pages 74-84, 2016, Elsevier. [Ferme et al., 2016b] - Presentation: https://www.slideshare.net/vincenzoferme/a-containercentric-methodology-for-benchmarking-workflow-management-systems, last visited on February 7, 2021

M. Skouradaki, **V. Ferme**, C. Pautasso, F. Leymann, and A. van Hoorn. **Micro-Benchmarking BPMN 2.0 Workflow Management Systems with Workflow Patterns**. *In Proc. of the 28th International Conference on Advanced Information Systems Engineering (CAiSE '16)*, pages 261-272, 2016, ACM. [Ferme and Pautasso, 2018] - Presentation: https://www.slideshare.net/MarigiannaSkouradaki/micro-benchmarking-wfms-with-workflow-patterns, last visited on February 7, 2021

C. Jürgen, **V. Ferme**, and H.C. Gall. **Using Docker Containers to Improve Reproducibility in Software and Web Engineering Research**. *In Proc. of the 16th International Conference on Web Engineering (ICWE '16)*, pages 609-612, 2016, Springer. [Cito et al., 2016] - Presentation: https://www.slideshare.net/vincenzoferme/using-docker-containers-to-improve-reproducibility-in-software-and-web-engineering, last visited on February 7, 2021

V. Ferme, A. Ivanchikj, and C. Pautasso. **Estimating the Cost for Executing Business Processes in the Cloud**. *In Proc. of the 14th International Conference on Business Process Management (BPM Forum '16)*, pages 72-88, 2016, Springer. [Ferme et al., 2016a] - Presentation: https://www.slideshare.net/vincenzoferme/estimating-the-cost-for-executing-business-processes-in-the-cloud, last visited on February 7, 2021

(To be continued)

Publications

Contributions to international conferences - Posters

V. Ferme, and C. Pautasso. **Integrating Faban with Docker for Performance Benchmarking**. *In Proc. of the 7th ACM/SPEC International Conference on Performance Engineering (ICPE '16)*, pages 261-272, 2018, ACM. [Ferme and Pautasso, 2018]

Outreach activities - Invited Speaker

V. Ferme. **Towards a Benchmark for BPMN Engines**. *1st International Workshop on Performance and Conformance of Workflow Engines*, 06.09.2016. - Presentation: https://www.slideshare.net/vincenzoferme/workflow-engine-performance-benchmarking-with-benchflow, last visited on February 7, 2021

Outreach activities - Invited Tutorials

Jürgen Cito, **V. Ferme**, and Harald C. Gall. **Using Docker Containers to Improve Reproducibility in Software and Web Engineering**. *16th International Conference on Web Engineering (ICWE '16) and University of Milano-Bicocca*, 2016. - Presentation: https://www.slideshare.net/vincenzoferme/using-docker-containers-to-improve-reproducibility-in-software-and-web-engineering, last visited on February 7, 2021

2015

Peer-reviewed conference proceedings - Conference proceedings

C. Pautasso, **V. Ferme**, D. Roller, F. Leymann, and M. Skouradaki. **Towards workflow benchmarking: Open research challenges**. *In Proc. of the 16th conference on Database Systems for Business, Technology, and Web (BTW 2015)*, pages 1-20, 2016, BTW. [Pautasso et al., 2015] - Presentation: https://www.slideshare.net/MarigiannaSkouradaki/benchmarking-workflow-engines-open-research-challenges-btw-2015, last visited on February 7, 2021

M. Skouradaki, D. H. Roller, L. Frank, **V. Ferme**, and C. Pautasso. **On the Road to Benchmarking BPMN 2.0 Workflow Engines**. *In Proc. of the 6th ACM/SPEC International Conference on Performance Engineering (ICPE '15)*, pages 301-304, 2015, ACM. [Skouradaki et al., 2015b] - Presentation: https://www.slideshare.net/vincenzoferme/on-the-road-to-benchmarking-bpmn-20-workflow-engines, last visited on February 7, 2021

M. Skouradaki, **V. Ferme**, F. Leymann, C. Pautasso, and D. H. Roller. **"BPELanon": Protect business processes on the Cloud**. *In Proc. of the 5th International Conference on Cloud Computing and Service Science (CLOSER 2015)*, pages 241-250, 2015, Elsevier. [Skouradaki et al., 2015a]

(To be continued)

	Publications
2014	**V. Ferme**, A. Ivanchikj, and C. Pautasso. **A Framework for Benchmarking BPMN 2.0 Workflow Management Systems**. *In Proc. of the 13th International Conference on Business Process Management (BPM '15)*, pages 251-259, 2015, Springer. [Ferme et al., 2015] - Presentation: https://www.slideshare.net/vincenzoferme/benchflow-a-framework-for-benchmarking-bpmn-20-workflow-management-systems, last visited on February 7, 2021 **Contributions to international conferences - Demos** A. Ivanchikj, **V. Ferme**, and C. Pautasso. **BPMeter: Web Service and Application for Static Analysis of BPMN 2.0 Collections**. *In Proc. of the 13th International Conference on Business Process Management [Demo] (BPM '15)*, pages 30-34, 2015, Springer. [Ivanchikj et al., 2015] - Screencast: https://www.youtube.com/watch?v=NaQ79VHoOgE, last visited on February 7, 2021 **Outreach activities - Invited Speaker** **V. Ferme**. **Towards a Benchmark for BPMN Engines**. *TIM Solutions (Company), Munich, Germany*, 03.11.2015. - Presentation: https://www.slideshare.net/vincenzoferme/towards-a-benchmark-for-bpmn-engines, last visited on February 7, 2021 **V. Ferme**, and C. Pautasso. **Towards a Benchmark for BPMN Engines**. *W-JAX Conference, Munich, Germany*, 04.11.2015. - Presentation: https://www.slideshare.net/vincenzoferme/towards-a-benchmark-for-bpmn-engines, last visited on February 7, 2021 **Other artifacts with documented use** **BPMeter**. http://benchflow.inf.usi.ch/bpmeter, 2015. - Screencast: https://www.youtube.com/watch?v=NaQ79VHoOgE, last visited on February 7, 2021
2014	**Peer-reviewed conference proceedings - Workshop proceedings** M. Skouradaki, D. H. Roller, F. Leymann, **V. Ferme**, and C. Pautasso. **Technical open challenges on benchmarking workflow management systems**. *In Proc. of the 2014 Symposium on Software Performance (SSP 2014)*, pages 105-112, 2014, Symposium on Software Performance. [Skouradaki et al., 2014]

Table 1.2. Publications Related to "WfMSs Performance Benchmarking"

In Tab. 1.2 we present an overview of the main contributions in the area of

Workflow Management Systems performance benchmarking. I contributed, in collaboration with colleagues at the USI, and the University of Stuttgart, to the development of the first set of benchmarks for benchmarking the performance of WfMSs executing the BPMN 2.0 standard. My main contributions have been the definition of a methodology and a pattern language for benchmarking such middleware, and the development of an open-source framework for performance benchmarking such middleware. I then contributed work on assessing the performance of different WfMSs and different versions of the same WfMSs.

	Publications
2020	**Peer-reviewed journal articles** A. Avritzer, **V. Ferme**, A. Janes, B. Russo, A. van Hoorn, H. Schulz, D. Sadoc Menaschè, and V. Rufino. **Scalability Assessment of Microservice Architecture Deployment Configurations: A Domain-based Approach Leveraging Operational Profiles and Load Tests**. *Journal of Systems and Software (JSS)*, volume 165 (1), pages 1-16, 2020, Elsevier. [Avritzer et al., 2020] V. Rufino, M. Nogueira, A. Avritzer, D. Sadoc Menasché, B. Russo, A. Janes, **V. Ferme**, A. van Hoorn, H. Schulz, and C. Lima. **Improving Predictability of User-Affecting Metrics to Support Anomaly Detection in Cloud Services**. *IEEE Access*, volume 8, pages 198152-198167, 2020, IEEE. [Rufino et al., 2020]
2019	**Peer-reviewed conference proceedings - Conference proceedings** D. Okanović, A. van Hoorn, C. Zorn, F. Beck, **V. Ferme**, and J. Walter. **Concern-driven Reporting of Software Performance Analysis Results**. *In Proc. of the 10th ACM/SPEC International Conference on Performance Engineering (ICPE '19)*, pages 1-4, 2019, ACM. [Okanović et al., 2019] - Presentation: https://continuity-project.github.io/files/20190410-ICPE-ConcernDriven-16zu10-handout.pdf, last visited on February 7, 2021 A. Avritzer, D. Sadoc Menasché, V. Rufino, B. Russo, A. Janes, **V. Ferme**, A. van Hoorn, and H. Schulz. **PPTAM: Production and Performance Testing Based Application Monitoring**. *In Proc. of the 10th ACM/SPEC International Conference on Performance Engineering (ICPE '19)*, pages 39-40, 2019, ACM. [Avritzer et al., 2019]

(To be continued)

Publications

C-P. Bezemer, S. Eismann, **V. Ferme**, J. Grohmann, R. Heinrich, P. Jamshidi, W. Shang, A. van Hoorn, M. Villavicencio, J. Walter, and F. Willnecker. **How is Performance Addressed in DevOps? A Survey on Industrial Practices**. *In Proc. of the 10th ACM/SPEC International Conference on Performance Engineering (ICPE '19)*, pages 45-50, 2019, ACM. [Bezemer et al., 2019] - Presentation: https://continuity-project.github.io/files/20190409-ICPE-Industry-Survey-ICPE.pdf, last visited on February 7, 2021

H. Schulz, D. Okanović, A. van Hoorn, **V. Ferme**, and C. Pautasso. **Behavior-driven Load Testing Using Contextual Knowledge - Approach and Experiences**. *In Proc. of the 10th ACM/SPEC International Conference on Performance Engineering (ICPE '19)*, pages 265-272, 2019, ACM. [Schulz et al., 2019] - Presentation: http://design.inf.usi.ch/sites/default/files/talks/20190410-ICPE-BehaviorDriven-16zu10-handout.pdf, last visited on February 7, 2021

Other artifacts with documented use

PPTAM. https://github.com/pptam/pptam-tool, 2019.

2018

Peer-reviewed conference proceedings - Conference proceedings

V. Ferme, and C. Pautasso. **A Declarative Approach for Performance Tests Execution in Continuous Software Development Environments**. *In Proc. of the 9th ACM/SPEC International Conference on Performance Engineering (ICPE '18)*, pages 261-272, 2018, ACM. [Ferme and Pautasso, 2018] - Presentation: https://www.slideshare.net/vincenzoferme/a-declarative-approach-for-performance-tests-execution-in-continuous-software-development-environments, last visited on February 7, 2021

F. Klinaku, and **V. Ferme**. **Towards Generating Elastic Microservices: A Declarative Specification for Consistent Elasticity Configurations**. *In Proc. of the 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA '18)*, pages 510-513, 2018, IEEE. [Klinaku and Ferme, 2018]

A. Avritzer, **V. Ferme**, A. Janes, B. Russo, H. Schulz, and A. van Hoorn. **A Quantitative Approach for the Assessment of Microservice Architecture Deployment Alternatives by Automated Performance Testing**. *In Proc. of the 12th European Conference on Software Architecture (ECSA '18)*, pages 159-174, 2018, Spinger. [Avritzer et al., 2018]

(To be continued)

	Publications
2017	**Research Projects** **V. Ferme**. **Declarative Continuous Performance Testing for Microservices in DevOps**. *Awarded by: Swiss National Science Foundation (Doc.Mobility Program)*, 01.02.2018 - 31.07.2018. - Project: http://p3.snf.ch/project-178653, last visited on February 7, 2021 A. van Hoorn, **V. Ferme**, and H. Schulz. **DevOps-oriented Declarative Performance Testing for Microservices**. *Awarded by: HPI Future SOC Lab*, 01.05.2018 - 31.07.2018. - Project: https://hpi.de/forschung/future-soc-lab-service-oriented-computing/devops-oriented-declarative-load-testing-for-microservices.html, last visited on February 7, 2021 **Outreach activities - Invited Speaker** **V. Ferme**. **Continuous Performance Testing for Microservices**. *HPI Future SOC - Lab Day (Spring 2018)*, 17.04.2018. - Presentation: https://www.slideshare.net/vincenzoferme/continuous-performance-testing-for-microservices, Recording: https://www.tele-task.de/lecture/video/6774/, last visited on February 7, 2021
	Peer-reviewed conference proceedings - Workshop proceedings **V. Ferme**, and C. Pautasso. **Towards Holistic Continuous Software Performance Assessment**. *In Proc. of the 3rd International Workshop on Quality Aware DevOps (QUDOS '17)*, pages 159-164, 2017, ACM. [Ferme and Pautasso, 2017] - Presentation: https://www.slideshare.net/vincenzoferme/towards-holistic-continuous-software-performance-assessment, last visited on February 7, 2021 **Outreach activities - Invited Speaker** **V. Ferme**. **BenchFlow: A Platform for End-to-end Automation of Performance Testing and Analysis**. *SPEC RG DevOps Meeting*, 12.05.2017. - Presentation: https://www.slideshare.net/vincenzoferme/benchflow-a-platform-for-endtoend-automation-of-performance-testing-and-analysis, last visited on February 7, 2021 **Other artifacts with documented use** **BenchFlow Framework**. https://github.com/benchflow/benchflow, 2016-2020.

Table 1.3. Publications Related to "Performance Testing Automation (Declarative and not)"

In Tab. 1.3 we present an overview of the main contributions in the area of performance testing automation, declarative, and not. As part of the effort on benchmarking WfMSs, I identified some drawbacks of state-of-the-art performance benchmarking solutions, when it comes to automation and integration of the same in CSDL. The identified drawbacks led us to the identification of challenges for performance testing automation in [Ferme and Pautasso, 2017] and to a DSL and framework for declarative performance testing automation published in [Ferme and Pautasso, 2018]. We relied on the proposed declarative and goal-driven approach for performance testing automation for contributing multiple works in collaboration with other researchers as reported in the referenced table.

During my PhD-related research career, I also had the opportunity to serve in the committee or as a reviewer of different conferences, workshops, and journals, assisting in teaching Bachelor and Master course, supervise or co-supervise Bachelor and Master theses, and attend PhD courses.

I served as part of the program committee of the following conferences or workshops: PEaCE 2016, MTD 2017, ICPE 2018 - Artefact Evaluation Track. I served as a reviewer for the following conferences, workshops, or journals: EASE 2014, WeTSOM 2014, IEEE Transactions on Services Computing, BPM-WS 2015, IEEE Cloud Computing, Software Engineering Journal. I served as a teaching assistant in five Bachelor courses (Human-Computer Interaction Atelier in 2015, Software Atelier 3: The Web in 2014, 2015, 2016, and 2017) and two Master courses (Business Process Modeling, Management and Mining in 2016, and 2017). I supervised or co-supervised six successful Master theses and three successful Bachelor theses. The supervised/co-supervised Master theses are:

1) *Characterising Representative Models for BPMN 2.0 Workflow Engine Performance Evaluation.* Student (University, Date): Ana Ivanchikj (USI, 05.09.2014);

2) *Performance Measurement of Heterogeneous Workflow Engines.* Student (University, Date): Marco Argenti (USI and Politecnico di Milano, 11.09.2015);

3) *The BenchFlow Framework for Flexible Performance Data Collection and Analysis.* Student (University, Date): Gabriele Cerfoglio (USI, 09.09.2016);

4) *The BenchFlow Framework for Automated Performance Experiments Execution on Heterogeneous Middleware Systems.* Student (University, Date): Simone D'Avico (USI, 09.09.2016);

5) *Automating Goal-Driven Performance Tests in BenchFlow.* Student (University, Date): Jesper Findahl (USI, 06.09.2017);

6) *Declarative User Experience Regression Analysis in Continuous Performance Engineering.* Student (University, Date): Manuel Palenga (University of Stuttgart, Fall 2018).

The supervised/co-supervised Bachelor theses are:

1) *Concern-driven Reporting of Declarative Performance Analysis Results Using Natural Language and Visualization.* Student (University, Date): Christoph Zorn (University of Stuttgart, Fall 2018);

2) *Implementation of a security vulnerabilities aggregator for Cloud-native environments.* Student (University, Date): Francesco Berla (USI, June, 2020);

3) *Aggregating Kubernetes Resource Metrics.* Student (University, Date): Marco Nivini (USI, June, 2020).

Chapter 2

Foundations of Performance Testing Automation

In this chapter, we present an overview of performance testing automation and its challenges as standalone activity as well as when integrated as part of CSDL. We look both at academic literature and industry literature, and state-of-the-art solutions.

2.1 Performance Testing Fundamentals

"Performance testing is an exercise in which a system is subject to loads in a controller manner. Measurements of resource usage and system performance are taken during the performance test for subsequent analysis." [Bondi, 2014] Performance testing includes performance test planning, performance test definition, workload modeling, performance test execution, performance result analysis, and report redaction [Bondi, 2014]. Performance testing is part of the performance engineering activities executed to guarantee the performance of a software system. Other activities part of performance engineering are the collection of performance requirements, performance risks assessment, performance debugging, performance improvements of the system, capacity management, and planning [Bondi, 2014]. Performance testing is fundamental to guarantee the system is developed according to performance requirements and to isolate and expose the resource or resources that limits the system scalability. This is important to guarantee a correct deployment in production, and proper capacity management and planning. The performance testing process iterative, the same way software development is an iterative process nowadays. After the initial assessment of performance and resources limiting the scalabil-

ity, allowing the definition of the first version of the system's test suite, the software is subject to change in functionalities, as well as the performance requirements are subject to change according to the behavior and the number of the end-users. It is important to constantly update the test suite, the test parameters, and the kinds of defined and executed tests to identify new and upcoming possible bottlenecks, even if there's no need to eliminate them in the current system version. Doing so will save valuable time as workload increases.

Guaranteeing the possibility to define and execute performance tests set numerous challenges. Performance tests have to be defined to properly verify the performance requirements, and different test types have to be executed. Automation of such tests is very important to successfully follow along with iterative development processes. Thus, also the configuration of automation needed for test execution is very important in this context. This means deployment of the system, deployment of the testing infrastructure, data collection, metrics computation, and results analysis has to be automated as well. Given the software is expected to change over time, as well as its APIs, it is also important to facilitate updates and reuse of test definitions and adaptation to the different changes. In this work, we focus on the test specification and execution automation challenges.

2.2 Different Kinds of Performance Tests

Testing systems performance requires many different test types, validating different performance requirements. Different test types are characterized by different load functions, different kind of interactions with the system as well as different configurations for the system's services. Some examples of performance test types are: 1) the load test, issuing a predefined load on a given system's configuration to assess the behavior of the system; 2) the configuration test, issuing a predefined load on a changing system configuration to identify the best configuration for the issued load; 3) the acceptance test, issuing a predefined load on a fixed system's configuration and validating the system passes defined acceptance criteria; 4) the regression test, issuing a predefined load on a stable system's configuration and validating the new version of the systems does not deviate more than a given threshold from the previous version.

Test execution might have different scopes as well. Tests can be executed to validate performance requirements, but also for exploring or profiling the system's performance, as well as for debugging the system after identifying performance bottlenecks. Tests executed to validate the performance require-

ments requires proper automation so they can be executed multiple times over the development of the system. Exploring system performance is usually done for setting performance requirements, or for exploring how the system behaves after applying performance-impacting changes. Another case in which exploration testing is performed is when a system is going to be deployed on a new deployment environment, and the performance behavior of the system has to be assessed in such a new environment. A profiler is usually used during the development of a software system and monitors resource usage as well as the call tree among the system's functions for computing the time spent in each of the functions. Profiling techniques are also applied when performance testing is executed to debug the system's performance.

Different test types and scopes introduce different challenges on test specification and automation. Defining configuration tests, requires the possibility to model a performance space of possible configurations and automatically configuring the SUT for the different configurations, as well as deploying data collection services. Executing performance regression tests requires access to previous execution results, to compare the performance with previous SUT's versions. Performing a test execution with the scope of profiling or debugging the system, requires the deployment of additional services next to the SUT to enable the profiling and debugging capabilities.

2.3 Automation in Performance Testing

Automation is fundamental to guarantee proper and scalable performance test execution. As illustrated in previous sections, executing performance tests sets multiple challenges, thus automating all the automatable activities is fundamental to guarantee the challenges can be successfully faced.

2.3.1 Main Challenges in Automating Performance Testing

Automating performance test execution requires to automate the configuration of the test infrastructure, the configuration of the SUT as well as its deployment on dedicated servers, the monitoring of the healthiness of test infrastructure and SUT during the test execution, performance data collection, metrics computation and automated evaluation about when to interrupt the test execution as well as in evaluating the quality of the SUT's performance after test execution.

The mentioned challenges require test specification languages and execution

frameworks embedding the performance test automation process configuration as part of the performance test specification. The users have to be able to specify the tests, as well as criteria to decide when to interrupt the test execution because of unexpected issues, while the execution framework must be able to continuously assess the healthiness of the test executing infrastructure and the SUT's services to guarantee correct error handling as well as ensuring the quality of collected performance data and computed metrics.

The SUT has to be threatened as a first-class citizen of the test specification language, so that configuration of the same can be controlled for complex test specifications. Data collection has to be ensured, as well as the identification of the different phases of execution of a performance test to validate the quality of the collected performance data. Metrics computed on top of collected performance data have to be validated as well, thus providing statistics and statistical tests along with actual metrics is important to validate the quality of obtained results. Metrics, statistics, and statistical tests can be used as a reference to define quality criteria to evaluate whether the system respects defined performance requirements or not.

2.3.2 Performance Testing Automation in DevOps

When integrating performance testing automation with iterative processes such as DevOps, the entire infrastructure for tests have to be handled automatically, as well as proper integration between development lifecycles and operations lifecycles have to be provided. When scheduling tests as part of an automated software build pipeline, it is important to provide means to select tests and moments of such automated software build pipelines in which to execute specific tests. Given the continuous execution of tests, and velocity requirements of DevOps processes, techniques to reduce the number of executed performance tests, a fine-grain selection of tests for example based on the risk of executing or not executing it, as well as optimization and re-use of previously accumulated performance knowledge is very important.

Proper integration with DevOps and DevOps-like processes requires the definition of test suites to be executed as part of software build and deployment pipelines. Such test suites define a set of tests to be executed and identify the moments in the development lifecycle of the system where the execution is expected to happens. It is also important to provide means to define the overall result of a test suite execution, so pipelines can decide actions to be taken according to the obtained results, the types of executed tests, and the moment in the system lifecycle in which the execution happened. state-of-the-art tools

for designing and implementing complex DevOps processes, such as GitLab[1], recently started to offer integration of performance test execution as part of software build and delivery processes. The integration is still preliminary, and limited support for the overall performance test execution activities is provided. Nonetheless, such widely-used tools acknowledge the importance of integrating performance tests as part of the lifecycles and facilitate the integration of performance requirements as part of the system backlog for better scheduling performance-related activities during the software development iterations. This is very important because "the keys to fully integrating performance testing into an Agile process are team-wide collaboration, effective communication, a commitment to adding value to the project with every task, and the flexibility to change focus"[2].

2.4 Concluding Remarks

In this chapter, we briefly introduce software performance testing. We then illustrate the different kinds of tests to be defined for successfully assessing performance requirements. We present the challenges one has to face when defining and implementing performance tests, as well as we discuss different scopes for which performance test are defined and executed. We then illustrate the importance of automation for successful performance test execution along with the software development, the multiple challenges in supporting proper and complete performance test specification and automation, and additional challenges introduced by the automated and continuous execution of performance tests as part of the CSDL. The main challenges we identify are related to performance testing automation and includes the support for performance test automation process configuration and integration in CSDL, other than actual performance tests definition as part of language and tools one relies on to automate the performance tests. This is important to provide a complete solution, and for sharing performance tests across the different teams contributing to the development of the system. By defining a complete definition, and configuring the test automation process, different people can execute the same or similar tests. The integration with CSDL enables the shift-left in performance tests execution, guaranteeing a complete integration with DevOps and DevOps-like practices.

[1] https://about.gitlab.com/, last visited on February 7, 2021
[2] http://www.perftestplus.com/, last visited on February 7, 2021

Chapter 3

State-of-the-Art

In this chapter, we present the state-of-the-art related to the context of our thesis. We briefly overview different areas of performance testing literature, and then we focus on performance testing methods and techniques applied in DevOps and alongside the CSDL. We present the main performance testing techniques for the main target system of our approach, namely RESTful Web services and BPMN 2.0 WfMSs deployed as Docker services. The last part of the state-of-the-art overview focuses on DPE and its contribution to performance testing, performance testing domain specific languages, and performance testing automation tools both from academia and industry.

3.1 Bird's-eye view of Performance Engineering and Performance Testing Literature

"Software Performance Engineering (SPE) is a systematic, quantitative approach to the cost-effective development of software systems to meet performance requirements" [Smith and Williams, 2001]. Performance engineering is a widely discussed topic in academia, having dedicated conferences such as International Conference on Performance Engineering (ICPE) and dedicated tracks in prominent software engineering conferences, such as International Conference on Software Engineering. The same is true in industry, with many companies offering tools and consultancy services for performance test definition and execution, as well as guidelines and practices for effective performance test execution. Nonetheless, performance engineering activities are not commonly executed as part of software development [Leitner and Bezemer, 2017], mainly due to the inherent complexity of defining and executing performance

tests. Thus in recent years, techniques to improve the integration of performance engineering activities with modern software development processes have been proposed, as well as new paradigms such as DPE have been conceived.

Different theories and techniques have been developed to introduce performance engineering in different moments of the software development lifecycle, and the main ones are model-based performance engineering and experiment-based performance engineering techniques [Bondi, 2014]. Performance testing belongs mainly to the latter technique, enabling the measurement of the system's performance under different load conditions, configurations, and deployment contexts.

Some of the proposed performance engineering techniques can be applied at software design time [Balsamo et al., 2004; Molyneaux, 2014; Mani et al., 2013; Woodside et al., 2013; D'Ambrogio et al., 2001; Alhaj, 2014], for example, performance modeling techniques [Cortellessa et al., 2011; Nambiar et al., 2016]. The techniques applied at software design time do not require an implemented software but require performance experts for their definition and execution because modeling techniques are usually complex to be adopted and difficult to tune to the actual use case to which they are applied to [Nambiar et al., 2016]. Modeling approaches can also be applied in different moments of the software development lifecycle when software has been developed and engineers plan to improve its performance [Nambiar et al., 2016; Arcelli et al., 2015; Cortellessa et al., 2011], in which case it is possible to use actual performance measurement data to tune models' parameters [Westermann, 2014; Levy and Steinberg, 2011]. The same techniques can also be applied when the system is deployed in production and the engineers have access to operations data [Brebner, 2016; Ardagna et al., 2010; Pachidi and Spruit, 2015]. Performance modeling techniques relying on measurement data are often focused on specific programming languages and deployment architectures [Cortellessa et al., 2011], because they require fine-grain data (e.g., collected through software instrumentation [Kempf et al., 2008; Horký et al., 2016; Kuperberg, 2010]) to enhance model prediction precision [Kuperberg, 2010]. The main advantage of model-based solutions, often once the model has been defined and trained/-parameterized with fine-grain real data is that they can be used for fast and accurate predictions of the application performance for new scenarios [Marin et al., 2014], on new hardware [Herbold, 2015; Marin and Mellor-Crummey, 2004; Zheng et al., 2015; Yang et al., 2005], in presence of unstable performance behavior of the deployment servers [Jimenez et al., 2016] and even across different hardware [Eden, 2011; Zheng et al., 2015; Jimenez et al., 2016; Rudolph and Stitt, 2015; Marin and Mellor-Crummey, 2004], especially if interested

in relative performance differences between two different servers [Yang et al., 2005] and not cross-server absolute performance prediction.

Complementary and/or alternatively to software performance modeling practices, many techniques for experiments-based software performance have been developed [Jiang and Hassan, 2015]. Some of these techniques are also used to fine-tune performance models, that are then used for performance predictions [Westermann, 2014; Marin et al., 2014; Brebner, 2016]. Experiments based techniques enable the users to obtain performance insights on developed and evolving software, by repeated performance experiments targeting the system during its evolution. Experiments based techniques are often offered to rely on tools automating the performance testing processes and are more and more exposed to the users by relying on a DSL simplifying the definition of performance experiments. Pure experiment-based techniques could require many experiments to collect performance knowledge about the SUT, that is the reason why they are often coupled with model-based techniques [Westermann, 2014], in a hybrid model- and experiment-based techniques. Experiments based techniques are defined according to different requirements. Exploratory performance testing is performed to learn about the performance of the system or third-party dependencies [Podelko, 2008; Hauck et al., 2013]. Load, Stress, Spike performance tests are defined and executed to test real-world systems to ensure they can function correctly under load [Jiang and Hassan, 2015; Chen et al., 2017]. The types of executed tests also depend on the system architecture and deployment context. Modern system architectures, for example, Microservice-based architectures [Lewis and Fowler, 2014], introduce many challenges in defining reliable performance tests and collecting reliable performance results. This is mainly because establishing a performance baseline is challenging in Microservice applications [Eismann et al., 2020], and a baseline is needed for effective performance testing. Defined and executed performance tests evolved with system deployment infrastructure, and the more and more variance in performance demonstrated by modern deployment infrastructures, for example, the Cloud [Leitner and Cito, 2016]. Due to the instability of performance behavior demonstrated by modern deployment infrastructure, performance tests are also defined to explore and learn about infrastructure performance [Jayasinghe et al., 2014]. Additionally, techniques for improving the reliability of performance results in such deployment environments have been proposed over time [Arif et al., 2018; Bachiega et al., 2018]. Although some deployment environments demonstrate variances impacting performance test result reliability [Barik et al., 2016], it is still important to test the performance of the systems in such environments, being the environment in which the

system is going to be deployed in production. To cope with the complexity of modern deployment environments and systems, artificial intelligence and machine learning have been applied to performance management and performance testing. Solutions guiding the performance testing process [Helali Moghadam et al., 2019], approaches for performance test generation [Grano et al., 2019], as well as solutions to optimize the performance test execution process [Abedi and Brecht, 2017] and to capture performance knowledge to advise developers about possible performance improvements have been proposed [Huck et al., 2008].

Development processes also impacted the approach to performance testing, especially when software development moved from a Waterfall approach to a more iterative and with continuous deployment to production environments approach (e.g., when applying DevOps). Modern iterative development and release processes introduce more uncertainty on the possibility to control software performance [Trubiani et al., 2018], thus requiring new techniques for performance test execution, that now has to happen continuously. More details about how iterative processes impact the definition and execution of performance tests are discussed in Sect. 3.2.3, and in the remainder of this work. To bring performance awareness closer to developers in the context of DevOps practices approaches to integrate performance feedback into the Integrated Development Environment (IDE) have been introduced [Vögele et al., 2014]. The integration of performance feedback in the IDE aims at bringing performance knowledge about the system in production to the developers directly in the environment where they are developing the software. To facilitate IT professionals to approach performance testing, something deemed more and more important when applying approaches continuously deploying iteratively improved versions of the software to production, tools, and techniques have been proposed to simplify performance tests definition by utilizing high-level DSL (refer to Sect. 3.6) or graphical interfaces, and frameworks to automate performance test scheduling and execution (refer to Sect. 3.7). Over time, the tools evolved also to be more and more integrated with CSDL, offering the opportunity to schedule performance tests execution from software build and deployment processes, e.g. relying on bots [Okanović et al., 2020] or on orchestration tools enabling definition of complex processes and interaction with performance testing tools [Dynatrace, 2019], as well as while the software is deployed in production [Brunnert and Krcmar, 2017].

Most recently, to improve on the support and accessibility of performance testing for professional figures involved in modern software development, DPE approaches have been defined and proposed. More details on DPE approaches

for performance testing definition and automation, and performance engineering activities are discussed in Sect. 3.5.

3.2 DevOps and Performance Engineering

In this section, we present what DevOps is, how it is related to CSDL, and how performance engineering activities, and especially performance testing are impacted by the characteristics of DevOps and CSDL.

3.2.1 What is "DevOps"

DevOps is a set of cultural approaches and processes aiming at coordinating software development, deployment, and operations as a uniform process bridging the often present gap between development and operations teams [Forsgren et al., 2018; Weber et al., 2015]. In most companies, the software is developed by development teams, and deployed and operated by operations teams. Both teams interact with the security team to include security guidelines in the followed processes. It is common to have the mentioned teams as separated silos, lacking communications, and knowledge transfer. The DevOps approach attempt to solve this separation, by asking different teams to collaborate towards the main company goal for IT, which is enabling business value. DevOps propose to do this by changing the culture of the company, fostering communication, responsibility-sharing, embracing failure, and proposing different team topologies [Skelton and Pais, 2019] suitable to introduce a new DevOps team as a facilitator of DevOps approaches including heavy automation, traceability, and pervasive measurements of processes efficiency and business value improvements [Bruel et al., 2019; Lwakatare et al., 2015]. The different teams' topologies are all characterized by the presence of professional IT practitioners from different areas of the company, having heterogeneous knowledge in different areas of IT and openness to collaboration, information sharing, and supporting colleagues. Typical of companies adopting DevOps is also the utilization of modern software deployment and operation platforms, such as Docker [Docker Inc., 2013] and Kubernetes [Kubernetes, 2014], designed to natively support automation and DevOps. More and more companies adopted DevOps practices over the years [Puppet, 2019; Google, 2019], and studies found DevOps practices are effective in improving IT ability to produce business value [Erich et al., 2017].

3.2.2 Continuous Software Development Lifecycle

Continuous software development, release, and iterative improvements of the same is an integral part of DevOps. We refer to this process as Continuous Software Development Lifecycle. CSDL strives to continuously integrate (Continuous Integration (CI)) [Duvall et al., 2007] and build the software developed in different development branches by different developers so that it is built, validated, tested, and packaged to be ready for deployment at any given point in time, enabling Continuous Delivery (CD) [Humble and Farley, 2010].

Developing software following a CSDL is important for embracing the main aspects of DevOps, namely sharing, automation, and measurement. This is made possible by the standardization of processes going from code commits to source code repositories by developers, up to the deployment of the software in production by DevOps or operations engineer, as well as operations and maintenance of the same. Standardized processes include contributions from the different teams involved in software development and secure and reliable operations, thus facilitating knowledge sharing. Such processes are mostly automated and measured in terms of the time it takes to bring new software versions to production, and other business-relevant metrics.

Adopting a CSDL introduces many challenges for performance testing execution and application of performance engineering activities. It also introduces opportunities, for example, relying on the standardization of processes typical of DevOps one can decide different moments of the CSDL in which to execute different types of tests and continuously improve performance knowledge about the developed application.

3.2.3 Performance Engineering and CSDL

Traditional software performance engineering practices are intrinsically slow. To carefully test the performance of a system and collect performance knowledge, it should be tested in a realistic environment, with realistic loads and data [Willnecker et al., 2016]. Additionally, performance engineering practices are also considered quite complex to be approached and executed in the right way, which is also made evident by the existence of dedicated professional figures, e.g., software performance engineers [Bosch, 2014]. When applying performance engineering to DevOps processes, the slowness, and the complexity of the same is an issue. Over time different approaches have been proposed and implemented to tackle the challenges of making it possible to apply performance engineering in a more efficient and user-friendly way. The challenges of

adapting and integrating performance practices in the CSDL are acknowledged by the scientific community [Brunnert et al., 2015; Walter et al., 2016] and by the industry [Reitbauer et al., 2015; Zalavadia, 2016].

The main challenges [Brunnert et al., 2015] in achieving a successful integration, are due to the characteristics of the context, e.g., timeliness to reach the deployment of a new incremental release, and the need for pervasive automation to achieve it. Other challenges are due to the professional figures that are involved, e.g., the need to simplify the way developers can obtain performance insights [Walter et al., 2016]. Any performance testing practice introduced in this context should enable and assist the different professional figures taking part in continuous software improvement [Fitzgerald and Stol, 2017] to implement and/or adopt performance-related activities as much as independently as possible from their performance expertise.

In DevOps processes, performance practices are pervasively integrated into many activities, because of the faster iteration cycles from development to deployment to production [Walter et al., 2016; Willnecker et al., 2016]. They cover from software design time, up to operations by relying on data and insights collected using Application Performance Management (APM)s [Ahmed et al., 2016; Kowall, 2016] and/or by deploying autonomic performance management agents monitoring the performance of the software in production and adapting parameters (e.g., configurations) to improve its performance for the currently experienced workload [Gambi et al., 2016; Lahmar et al., 2015; Ardagna et al., 2010]. The integration of performance testing, analysis, and engineering activities within the CSDL is a widely recognized challenge in the literature [Bezemer et al., 2019; Walter et al., 2016; SPEC, 2016; Brunnert et al., 2015; Casale, 2016; Gottesheim, 2015; Kroß et al., 2016; Pitakrat and Heinish, 2016]. CI lifecycle automation [CodeShip, 2016; Haghighatkhah et al., 2017; Elbaum et al., 2014; Humble and Farley, 2010] best practices organize tests in workflows, pipelines, and steps. It is recognized the complexity of performance testing, the complexity of integrating performance testing tools in CSDL, and the lack of native support for integration of performance tests in CI and CD tools limits the adoption of performance testing in CSDL [Yu et al., 2020].

As anticipated in Sect. 1.1, two research activities in the area, relevant for the work of this dissertation, are the DECLARE project[1], and the ContinuITy project[2]. The DECLARE project "envisions to reduce the current abstraction gap between the level on which performance-relevant concerns are formu-

[1] http://www.dfg-spp1593.de/declare, last visited on February 7, 2021
[2] https://continuity-project.github.io, last visited on February 7, 2021

lated and the level on which performance evaluations are actually executed", thus dealing with the challenges related to the heterogeneity in performance expertise of software practitioners. The DECLARE project focuses mainly on enabling the possibility of declaratively querying performance knowledge that has been collected and modeled by other systems and proposes languages and tools for specifying performance concerns and obtaining automated answers [Walter, 2018]. The ContinuITy project focuses on dealing with the challenges of continuously updating performance tests [Schulz et al., 2020], by leveraging performance knowledge of software systems collected and modeled from the software operating in production environments. In the context of the ContinuITy project, performance tests are considered as disposable artifacts, evolving according to the current and foreseen requirements of the system they target. By monitoring services in production environments, the project aims at collecting performance knowledge that can be then utilized to constantly update, in a semi-automated manner, the performance tests defined for a system. Those tests can be then continuously executed during the CSDL, by other frameworks, to validate the performance of the system over time.

Other than what covered by the two research projects mentioned above, the different research and practical directions on the topic cover the study of the possibility to integrate existent performance testing tools in the CSDL to enable continuous experimentation, embracing failure [Geiger et al., 2014], the use of performance unit tests [Horký et al., 2013; Reichelt and Scheller, 2015; Bulej et al., 2014; Horký et al., 2015; Bulej, 2016; Ding, 2019] and benchmarks applied continuously [Grambow et al., 2019] or to nightly builds [Waller et al., 2015; Bulej et al., 2005] for early performance regression detection, the use of continuously collected performance knowledge to optimize the set of executed performance tests to maximise failure detection while reducing the number of executed tests [Memon et al., 2017; Jaewon et al., 2018; Hashemian et al., 2017], the use of A/B testing (e.g., as done in Google Vizier [Golovin et al., 2017]) to evaluate performance changes between different versions of the same system, the use of live testing [Gerostathopoulos et al., 2016; Schermann et al., 2016; Weaveworks, 2018; Dynatrace, 2019] for incrementally roll-out a new version of a Microservice application according to its performance behavior, the use of modeling techniques [Dlugi et al., 2015] and machine learning [Simon, 2014] to reduce the time needed to gain performance insights on the developed software at any moment of the development lifecycle, and the use of operations performance data to automatically build application performance models dedicated to specific architectures and programming languages [Brebner, 2016].

Overall current approaches propose to integrate performance testing in dif-

ferent ways and moments in the continuous software improvements lifecycle. In our research we build on the performance testing literature, and in particular, the literature in the context of DevOps and CSDL, and we embrace the DPE proposition to provide languages and tools to help professionals working in continuously improving software, to define and execute performance tests as part of the continuous improvement activities.

3.3 Automation of Web Services Performance Testing

RESTful Web services are one of the two main target systems of our approach, in particular when deployed using Docker technology. The identification of this category of systems as a target of our approach is since many software systems are developed using Web technologies [SmartBear, 2020] and such systems are usually exposed by a potentially large number of users, both real and other systems, thus requiring careful performance assessment and iterative development.

The purpose of Web service performance testing is to check the reliability and the performance, of services APIs. Web services performance testing issues load to the services to check how they perform for single clients and how they scale as the number of clients accessing it increases. Testing of Web services enables the detection of performance issues, and automated approaches, in particular, help to efficiently repeat tests whenever needed.

Various approaches have been proposed for defining and executing performance testing of Web services. [Canfora and Di Penta, 2009; Vögele et al., 2014] discusses different techniques for testing systems SLAs and dependability relying on performance testing. The authors of the referenced survey and papers indicate the following techniques as the most commonly applied for performance testing of Web services:

a) generation of test input during performance tests using genetic algorithms, trying to maximize the number of SLA violation during tests, to improve the system and protect it from possible violations in production. The monitored properties are response time, throughput, and reliability;

b) fault injection during the interaction with the Web service APIs;

c) use of performance models to reduce the time to execute performance testing in large Web services deployment.

In [Jiang and Hassan, 2015] the authors overview the different techniques of approaching performance testing of large-scale software systems, mainly of Web services. Some of the discussed techniques are similar to the work in [Canfora and Di Penta, 2009; Vögele et al., 2014], for example, the need of defining tests injecting faults, and additional techniques are presented for the following concerns relevant for performance testing of Web services:

a) execution of different types of performance tests to answer to different performance concerns;

b) relying on production usage data from current versions deployed in production to improve the executed load tests, and issue a more realistic load during testing activities.

There are various Web services performance testing tools available implementing the discussed techniques, such as JMeter. Some tools are available on the market as open-source or as a commercial solution, others are research prototypes implementing state-of-the-art and advanced experimental techniques. We present an overview of the different tools for performance testing of Web services in Sect. 3.7, and we discuss how the approach we propose differentiate from the one proposed by the available tools.

3.4 Automation of Workflow Management Systems Performance Benchmarking

BPMN 2.0 WfMSs are the second main target system of our approach, in particular when deployed using Docker technology. The identification of this category of system stem from the initial contribution in the area of benchmarking BPMN 2.0 WfMSs presented in Sect. 1.5 and in Sect. 8.1, and from the fact BPMN 2.0 WfMSs usually handle interactions of many different real users and other systems, and require careful performance assessment.

The need for benchmarking WfMSs has been identified by many practitioners, and starting from 2013-2014 has been comprehensively tackled by the BenchFlow research project[3]. In 2000, [Gillmann et al., 2000] release a benchmark for comparing the performance of different commercial WfMSs. The

[3] http://design.inf.usi.ch/research/projects/benchflow, last visited on February 7, 2021

benchmark is built to measure the WfMSs' throughput, particularly focusing on the impact of the load on the underlying Database Management System (DBMS) the WfMSs relies on to store the state. Approximately ten years later, in 2010, SOABench [Bianculli et al., 2010b] defines one of the first performance comparisons for Business Process Execution Language (BPEL) WfMSs. SOABench characterizes the performance of BPEL WfMSs by the response time of process calls. A review [Röck et al., 2014] of the work published so far for benchmarking BPEL WfMSs, underlines the lack of an extensive evaluation of different WfMSs in terms of multiple workload and metrics, the ambiguous definition of the workload mix (i.e., the mix of process models executed during the performance tests) and the load functions, as well as the lack of narrowly focused metrics for characterizing WfMSs's performance, often characterized with standard performance engineering metrics.

Given the complexity of WfMSs, it is very important to automate as much as possible the entire performance testing processes. WfMSs come with many different configuration and deployment options, and many different kinds of process models can be executed by the same. Multiple repetitions of performance tests have to be performed to provide high-quality results, and especially in the case of benchmarking such systems, exploring the performance of the same with multiple tests, deploying different process models under different loads is fundamental. Many commercial and open-source performance measurement frameworks are available on the market [Molyneaux, 2014], some focusing on generic Web Applications (e.g., Faban [Faban, 2000], JMeter [JMeter, 1998]), and others purposefully built for performance testing middlewares [Bianculli et al., 2010b]. However, to the best of our knowledge, no ready-to-use solution for automated end-to-end performance testing BPMN 2.0 WfMSs was available before the one proposed as part of the BenchFlow project [Ferme et al., 2015].

3.5 Declarative Performance Engineering

DPE, part of which is also related to performance testing, has been presented as part of the DECLARE project [Walter et al., 2016] and analyzed more in-depth in [Walter, 2018]. DPE has been proposed as a new paradigm to facilitate access to performance engineering activities and to disseminate awareness about the need of applying performance engineering in different moments of the software development lifecycle. DPE focuses on allowing users to specify the facets of the system's performance they are interested in, without requiring complex

specification and implementation of the mechanism needed to answer to the requested performance intent. DPE indicates the complexity of performance engineering has to be abstracted away from users and codified in expert systems. Users need to be able to interact with such expert systems with abstract enough languages to state their performance intent and optionally control the processes followed to answer to such intent. They are not required to specify the details of how the process is implemented, executed, and how performance results are retrieved and analyzed.

Related to DPE, [Westermann, 2014] present the concept of goal-driven performance testing, mainly related to the smart exploration of the system's performance space defined by the combinations of multiple configuration options the systems accept. The performance queries the user can specify in the work by [Westermann, 2014] are mostly limited to configurations exploration. [Westermann, 2014] enable the performance space exploration providing a DSL and a runtime framework for automatic performance test execution. The main differences between the work by [Westermann, 2014] and our approach are the context of the application, and the focus of the declarative goal-driven definitions. In our approach, we focus on performance testing of container-packaged services within CSDL lifecycles, and we defined goal-driven definition open to answer different performance intents users might have, and not only related to configurations exploration. Other relevant work proposing both a declarative DSL and an automation framework, are Cloud Work Bench [Scheuner et al., 2014], Crawler [Cunha et al., 2013], CloudPerf [Michael et al., 2017], and Jagger [Dynamics, 2014], tools for benchmarking the performance of the solutions offered by Cloud providers. The first three tools propose a declarative DSL specifically tailored to Cloud benchmarking, thus targeting a different domain than the one proposed in the context of this work. Although Jagger's DSL allows users to declaratively specify the success criteria of tests, in our goal-oriented approach we additionally support describing the intent of performance tests. Other related works propose dedicated approaches to answer different kinds of performance intents, for example, capacity planning [Spinner et al., 2015], and performance optimization in the presence of constraints [Di Alesio et al., 2013], white and black box regression detection [Kalibera and Tůma, 2006], performance bugs identification and prediction [Tsakiltsidis et al., 2016], and characterize the performance of server infrastructures [Hauck et al., 2013] and of software components [Kolb et al., 2006]. We consider these techniques as declarative approaches to performance testing because the discussed solutions target specific performance goals to be answered in a (semi)-automated way. Some preliminary approaches related to DPE, and in particular to goal-driven

performance testing, are also present in commercial software. Load Runner by Microfocus[4] allows the definition of goal-oriented performance tests where the users state the goal, and the system takes care of handling the processes to reach such goal. Only basic goals are available to date, mainly related to testing the system capacity in terms of the number of supported users, given constraints in terms of maximum response time and other performance relevant metrics. A second relevant goal-oriented approach in the industry is the one adopted by Percona [Schwartz and Zaitsev, 2010] for their database performance optimization consultancy services. Percona experts propose a process where performance goals are made clear and reported before starting performance testing activities. Then performance tests are defined specifically to measure the current performance of the system, detect performance issues hindering to reach performance goals, and optimizing the system to achieve the stated goals. As it is for all abstraction processes, in DPE low-level control on performance engineering activities is limited, to favor the simplicity of the approach. Complexity can be codified as part of DPE approaches, but careful selection of what is exposed to the users and what is hidden has to be performed to guarantee the principle of DPE.

To the best of our knowledge, the state-of-the-art in DPE does not comprehensively cover performance testing specification and automation, and their integration within CSDL and DevOps. Work on the topic has been mainly presented and discussed in [Walter, 2018], where the author proposes a language for specifying performance intent and a platform able to decide about the best technique to answer the stated query. With our work we contribute towards filling a gap in the work by [Walter, 2018], mainly focusing on performance modeling, and contribute a solution allowing the users to declaratively specify and reuse goal-driven performance test specifications and automate their execution alongside the CSDL. Performance tests specification and execution is a needed capability when referring to DevOps contexts, where continuous improvements are made on the software, and it has to be continuously tested over its different versions. To the best of our knowledge no similar solutions exist.

[4] https://www.microfocus.com/en-us/products/loadrunner-enterprise/overview, last visited on February 7, 2021

3.6 Performance Testing Domain Specific Languages

Performance test specification requires languages and tools to enable users to describe their performance tests. Many different approaches have been proposed, and most of them are based on DSLs allowing to codify performance-related domain concepts. Some of the DSLs are also supported by user interfaces to facilitate the specification of complex tests.

DSLs are widely used by IT practitioners for different scopes [Fowler, 2010]. The design of Domain-Specific Languages is also studied in the literature [Fowler, 2010; Voelter et al., 2013; Frank, 2013; Karsai et al., 2014; Spinellis, 2001; van Deursen et al., 2000], with a large number of possible choices for language syntax and semantics. In the context of system testing, different DSLs have been proposed for both functional [Micallef and Colombo, 2015] and non-functional testing. Regarding performance testing related DSLs, the large number of possible choices led to different decision made by DSL designers. DSLs for performance testing are designed using YAML Aint Markup Language (YAML) (e.g., [Farcic, 2016; Blazemeter, 2014]), Unified Modeling Language (UML) (e.g., [Bernardino et al., 2016]), code (e.g., [Wang, 2006; Tolledo, 2014; Dynamics, 2014; JMeter, 1998]), or custom formalisms (e.g., [Bernardino et al., 2014; Spafford and Vetter, 2012]). All the mentioned performance testing focused DSLs support the definition of load functions, workloads, simulated users, and test data. Other DSLs also support testbed management [Wang, 2006; Westermann, 2014], analysis of client-side performance data for checking defined conditions [Blazemeter, 2014], and definition of configuration tests [Westermann, 2014; Brown and Hellerstein, 2004].

Declarative testing languages are available for defining functional test scenarios, especially when applying behavioral driven development [North, 2006]. One of the most widely accepted and used declarative language for declarative functional testing is Cucumber [Wynne and Hellesoy, 2012]. A declarative language enables the user to rely on shared domain knowledge to specify their request, without the need for specifying how to achieve an answer to the same. A declarative language can be considered executable if the process to achieve the answer to a specified request can be automated.

For what concerns declarative performance testing languages proposed in the literature, the most prominent are the ones proposed by [Westermann, 2014; Bernardino et al., 2014; Scheuner et al., 2014; Cunha et al., 2013; Michael et al., 2017; Blohm et al., 2016]. [Westermann, 2014] presents a declarative language for performance testing specification. The presented DSL allows for the specification of the test goal as well. The authors focus mainly on enabling the sup-

port for performance test specifications having the goal of exploring relatively large performance spaces and relying on statistical models to do so. Compared to the approach we propose in our work, the DSL in [Westermann, 2014] does not support a wide number of performance test types, focusing mainly on exploration testing, and misses the integration with CSDL. As part of the work, the author also proposes a framework for exploration performance testing automation. We took inspiration from the architecture proposed in [Westermann, 2014] for designing the framework proposed as part of this thesis. [Bernardino et al., 2014] present Canopus, a behavior-driven language for Web application performance test specification. The proposed language is graphically represented and serialized using Extensible Markup Language (XML). Compared to the DSL proposed in this work, Canopus does not allow the user to specify the performance test goals and has limited expressiveness in terms of different types of performance tests that can be specified, as well as it misses the support for the end-to-end configuration of the test automation process. [Scheuner et al., 2014; Cunha et al., 2013; Michael et al., 2017] each proposes a tool providing a declarative DSL and an automation solution dedicated to Public Cloud benchmarking, and in particular, to benchmark the performance of Virtual Machine (VM) on different Public Cloud providers. The proposed declarative DSLs are tailored to the Cloud domain, thus targeting a different domain than the one proposed in the context of this work and less expressive concerning the types of tests the user can define, other than missing integration with CSDL. The proposed frameworks are also exclusively dedicated to testing Cloud platforms. Nonetheless, we took inspiration from the architecture of the framework proposed in the mentioned work for designing our solution. [Blohm et al., 2016] present a declarative language for performance monitoring. It is related to performance testing, in the sense that the language is proposed to automatically configure performance data collectors deployed in production during live user utilization and obtain answers about performance behavior the application is experiencing.

Some solutions available on the market enable the integration of performance testing alongside the CSDL. The most relevant ones are the ones proposed by [Artillery, 2014; Blazemeter, 2014; GitLab, 2020]. All the mentioned approaches tackle the integration with CSDL by proposing YAML-based DSLs allowing the specification of tests and test suites to be executed, and enable the integration in different steps of the CSDL pipelines by relying on tools automating the test execution. [Artillery, 2014] support the users in the integration of performance testing by allowing the definition of simple load tests supporting multiple scenarios of interaction with the SUT, potentially having a different

weight. [Blazemeter, 2014] enables the specification of performance tests executed by many different execution frameworks (e.g., JMeter) and supports the integration with the CSDL by offering CI or CD tools specific plugins. The plugins enable the users to define different types of performance tests, configuring services to collect performance data both from the client- and the server-side and criteria to control when to terminate the test execution and how to assess the quality of the test result. [GitLab, 2020] recently[5] included the support and integration of load performance test execution as part of the Continuous Integration and Delivery (CICD) pipelines users can define on the tool. The integration enables the users to define load tests to be executed alongside the CSDL and the platform takes care of scheduling and executing such tests according to user-defined triggers. GitLab suggests to execute such tests in the context of a Pull Request, but it is not mandatory. Quality gates can be defined so that pipelines executing the tests can get feedback about the performance of the software and decide for a success/failure of the same accordingly. We base our proposal on such solutions available on the market and enhance them by enabling a declarative specification of test suites, as well as we empower the users in controlling test suites execution according to the different events generated during CSDL. This is something that to the best of our knowledge is not present in already available solutions, and we consider it important for better integration and control of performance testing execution as part of DevOps processes.

In this work, we build on the already proposed DSLs, and we provide an expressive DSL for declarative performance tests specification, where the users can state what they expect in terms of specifying the performance test goal, and can control the process to achieve an answer to such goal, executing a standalone performance test or integrating one or more performance tests as part of CSDL. The test execution process is not exposed to the user but implemented in a model-driven framework part of the proposed approach. With the proposed DSL users can describe and codify performance goals at a high-level of abstraction, and the model-driven framework takes care of converting the specification to executable code needed to execute performance tests and controlling the SUT lifecycle during test execution. To the best of our knowledge, no similar solutions are available. We consider the declarative DSL and related model-driven framework an important contribution for empowering users in executing performance tests.

[5]In July, 2020

3.7 Performance Testing Automation Tools

Performance engineering is assisted by many tools helping users in applying the different performance methods to different kinds of systems. We see tools spanning from the ones developed for assisting users with performance modeling activities [van Hoorn et al., 2012], to tools for monitoring and building performance models from performance data collected from software running in production [Willnecker et al., 2015], other than the so-called performance management platforms [Rigor, 2014; opsZero, LLC, 2016] offering an integrated approach.

A survey about performance testing tools and their characteristics has been published recently [Costa et al., 2020]. The survey presents a systematic literature review of different performance testing tools, mainly from academia and open-source ones, and their characteristics. For the performance testing automation tools overview in this section, we refer to such a survey, and select tools supporting the target systems of our work and/or enabling DPE. We further include additional tools: a) based on our experience in the performance testing domain over different years; b) from commercially available tool catalogs, relevant for our work; c) enabling declarative specification of performance tests; d) supporting performance testing in CSDL; e) supporting performance modeling techniques for performance exploration; f) targeting especially RESTful Web services and Workflow Management Systems; g) supporting Docker and DevOps-related deployment technologies.

In Tab. 3.1 we report the list of tools, classified into the following categories:

1) *General Purpose Performance Testing Tools (Open-source)* - Open-source tools for performance testing of Web services, especially targeting services' APIs;

2) *General Purpose Performance Testing Tools (Commercial)* - Commercial tools for performance testing of Web services, especially targeting services' APIs;

3) *Middleware Oriented Performance Testing Tools* - Tools for performance testing of middleware solutions, especially focusing on WfMSs, both supporting BPMN 2.0 and not supporting it;

4) *Cloud and/or Cloud-native Oriented Performance Testing Tools (Open-source and Commercial)* - Tools for performance testing Cloud-native architectures, such as Docker and DevOps-related deployment technologies, and Cloud deployment platforms.

For each tool, we report characteristics identified as relevant in the referenced survey as well as additional characteristics relevant in the scope of this work. The list of the reported characteristics for each of the tools is:

a) *Type* - whether the tool is open-source, commercial, or a research prototype/demo;

b) *Test Definition Language* - the languages or method by which the users can specify a performance test;

c) *CSDL Integration* - if and how the tool is integrated in CSDL;

d) *Metrics Computations* - whether or not the tool provides metrics computation capabilities, and whether they cover client- and server-side metrics.

The table cells mark whether the characteristic is present by using the ✓ symbol, or by reporting details about the present characteristics, or if the characteristic is not present by using the ✗ symbol.

In the following, we briefly discuss the performance testing automation tools in the different categories, and we highlight important characteristics of the tools relevant for this dissertation.

General Purpose Performance Testing Tools (Open-source) - Many open-source tools for performance testing are available on the market. Most of the available solutions enable an imperative approach to performance testing where the user is requested to model the performance test workload, configure systems to collect the performance data, and indicate the set of metrics she is interested in. The available tools support different protocols as target for the load test, from Hypertext Transfer Protocol (HTTP) to Java Message Service (JMS), Java Database Connectivity (JDBC), and Simple Object Access Protocol (SOAP). Not all of the available solutions support the distributed deployment of the load infrastructure, although the feature is becoming more and more popular due to the increasing demand for scalability in the number of simulated users. Most of the available open-source solutions do not support the health checking of the load infrastructure as well, an important feature to reduce the possible cause of issues impacting the test quality. An example of a solution supporting both distributed deployment and health checking of the load infrastructure is ElasTest [ElasTest, 2016].

Different tools adopt different approaches in supporting users in defining the tests. Few open-source tools offers also a Graphical User Interface (GUI) for supporting users to define performance tests. Most of the open-source tools we

identified enable test specifications using programming languages, mainly covering the workload modeling part of the performance test specification. Some tools, for example, Taurus [Blazemeter, 2014], enables specification of the performance test using the same language we use for the approach we propose, i.e. YAML. Other than enabling the specifications using programming or custom language, some tools also support the possibility to record interactions with services and build performance tests out of the recorded interactions log. This feature is particularly useful and relevant when testing applications presenting a GUI to the users.

For what concerns the support for integration with CSDL, few open-source tools enable such possibility. Although most of the tools accept the possibility to schedule a test via CLI or APIs, few of them offer capabilities enabling a native integration with CICD tools. Among the identified tools, the one natively integrating with CICD tools are Gatling [Gatling, 2011], Taurus [Blazemeter, 2014], Artillery [Shoreditch Ops, 2017], k6 [Load Impact AB, 2019], and ElasTest [ElasTest, 2016]. All of the mentioned tools support the specification of termination criteria, determining when and how to stop the test during its execution, and enable quality gates or checks, allowing users to specify successful and failure conditions for the test execution. Some of the tools support the specification of test suites or test collections. Over the years we saw an increasing number of tools supporting native integration with CICD tools, due to the increasing demand for such a feature in modern software development processes.

Concerning available performance metrics, all the tools support standard performance metrics looking at the number and velocity of interaction between the simulated users and the SUT, as well as metrics collected from the running services part of the SUT, e.g., resource utilization for Central Processing Unit (CPU). Some tools, especially research tools or tools developed out of research work in academia, enable support for more advanced quality control of performance tests execution and performance tests results' validation. BenchExec [BenchExec, 2015] relies on advanced interaction with the Linux kernel to properly isolate executed processes and measure their performance in terms of resource utilization. BenchExec can extract further statistical data from the output. Results from multiple runs can be combined into Comma-separated values (CSV) and interactive Hypertext Markup Language (HTML) tables. DataMill [Petkovich et al., 2015] enables rigorous performance evaluation, by controlling the behavior of the performance test infrastructure and the SUT deployment infrastructure during the entire performance test lifecycle. DataMill also codifies many best practices in performance data collection and

performance data analysis to ensure the collected data enable the computation of reliable performance metrics. The solution does not particularly focus on performance testing but enables the execution of performance tests as well. Ascar Pilot [Li et al., 2016] is designed to help users to obtain precise results out of their performance experiments. The tool enables the specification of precision constraints on the computer performance metrics and guarantees those constraints are respected or warn the user otherwise. It also helps in assessing the quality of performance results when compared across different executions of the same performance test targeting different versions of the SUT, to ensure the eventually identified differences are differences and not measurement errors. To achieve these goals Ascar Pilot automatically decides what statistical analysis method is suitable based on the system configuration, defined measuring requirements, and existing results while optimizing the performance test duration.

Some of the mentioned open-source tools, for example, k6 [Load Impact AB, 2019] and Gatling [Gatling, 2011], also offer a paid version of the tool. The main offered services for the paid versions across the tools are the availability of a Public Cloud infrastructure for simulating users issuing the load, advanced features for example integration with more third-party tools and advanced reporting, and dedicated support.

General Purpose Performance Testing Tools (Commercial) - As for open-source tools, also in the commercial space, many tools for performance testing are available on the market. The tools mostly support imperative approaches and enable performance testing targeting many different protocols. Commercial tools enable more advanced control of the SUT deployment management and lifecycle compared to the open-source tools. Management of SUT deployment as part of the activities performed during performance testing is available in some tools, e.g., NeoLoad [NeoLoad, 2016], but not in all the commercial solutions. Mostly all the available commercial tools check the healthiness of the test infrastructure, at least in one of the different pricing tiers they offer. Many tools also offer a Cloud-based tier, where the testing infrastructure is completely managed by the solution according to user-defined reservation criteria and the user does not need to take care of setting it up and guarantee test resources are available at any time.

Commercial tools enable test specifications mostly relying on advanced GUI. They also all offer the possibility to serialize the test specification in different output formats and some of the tools, e.g. NeoLoad [NeoLoad, 2016], also offer the possibility to specify tests using modeling languages. NeoLoad [NeoLoad, 2016] also offers a specification language including some declarative

elements for specifying SLAs of the SUT during the performance testing. All the commercial tools offer recording capabilities for the users wanting to record test scenarios while manually interacting with the SUT. This is a needed capability in enterprise based on our experience.

Almost all the commercial performance testing tools support the integration with CICD tools part of the CSDL. Based on our experience this is a more and more requested feature in Enterprise, thus commercial tools support this capability. Commercial tools support all the features mentioned for open-source tools, as well as advanced integration, mostly relying on CLI and APIs. Advanced features are available for enabling comparison of performance test results across different versions of the SUT, for enabling mocking of not available services and to isolate the test to specific services of the SUT. Some commercial tools, e.g. NeoLoad [NeoLoad, 2016], also support users with Artificial Intelligence (AI) algorithms analyzing test results to pinpoint bottlenecks and root-cause analysis of performance issues. NeoLoad [NeoLoad, 2016] also supports users in updating test scripts by maximizing the reuse of previously developed test scripts when updating services' APIs. Commercial tools often integrate with APM solutions, to enhance the performance data analysis capabilities and provide actionable insights to users.

Commercial solutions offer broader support for performance data collection and performance metrics computation when compared to open-source solutions. All the commercial tools offer client- and server-side metrics providing insights on all the aspects related to systems' performance. They also often provide dedicated metrics for different kinds of systems and protocols to better characterize their performance. In terms of results validation, also in the commercial tools advanced analysis of quality of results is not widely supported. Some tools, for example, RETiT [RETIT GmbH, 2020], BlazeMeter [Blazemeter, 2016], and NeoLoad [NeoLoad, 2016] support for advanced data analysis and validation, mainly relying on algorithms validating the performance test result.

Some of the commercial tools offer a complete platform for functional and non-functional testing of applications. Some examples are RETiT [RETIT GmbH, 2020], BlazeMeter [Blazemeter, 2016], and NeoLoad [NeoLoad, 2016], offering a complete solution allowing users also to reuse functional tests when defining performance tests, and provide a complete collaboration platform where different professional figures can collaborate to refine SLAs, develop tests and analyze results.

Middleware Oriented Performance Testing Tools - The two main identified tools in this context are SOABench [Bianculli et al., 2010b] and

SOArMetrics [Li et al., 2009]. The mentioned tools are particularly relevant for one of the main target systems for our approach, WfMSs. Both of the tools offer an integrated approach for performance testing definition and execution, automating the process as much as possible. Automation is fundamental in the context of middleware performance testing due to the many possible configuration options of such systems. SOABench [Bianculli et al., 2010b] and SOArMetrics [Li et al., 2009] define custom performance test definitions based on scripts, to also support SUT deployment and performance testing infrastructure management. They are not integrated with CSDL and support custom metrics specifically targeting WfMSs.

Cloud and/or Cloud-native Oriented Performance Testing Tools (Open-source and Commercial) - In the context of Cloud and/or Cloud-native performance testing, many dedicated tools have been developed both in academia and industry. Most of the proposed approaches are imperative, but some tools are supporting a declarative specification of performance tests for example Cloud Work Bench [Scheuner et al., 2014]. Tools dedicated to performance testing of Cloud technologies support for SUT provisioning/deployment and performance testing infrastructure provisioning and management. They are usually not integrated with CSDL, nor support the integration because performance testing activities they enable are usually scheduled independently of software development.

Different tools implement different approaches for test definition, from GUI-based definition to code-based definition. Some tools, for example, Cloud Work Bench [Scheuner et al., 2014], rely on standard infrastructure as code languages, such as Vagrant[6], for enabling the specification of the test and the provisioning of the infrastructure as part of the same specification.

Being mostly dedicated to test and assess the performance of VMs and containers provisioned on Public Clouds, the tools support for the integration with the main Public Cloud providers and collect performance metrics made available for such platforms. Performance metrics are mostly related to resource utilization of provisioned VMs and containers while executing given sets of workloads.

From the analysis of the state-of-the-art, we highlight a limited number of tools that enable a DPE approach for performance testing. These tools are available mainly for testing software in specific contexts, e.g. on the Public Cloud. The identified tools offer a declarative language for performance test definition, and some support specification of goal-oriented tests, and the

[6] https://www.vagrantup.com/, last visited on February 7, 2021

configuration of the performance test execution process. A limited number of open-source tools offer integration with the CSDL, while this feature is widely present in commercial tools. We enable the declarative integration of performance tests alongside the CSDL in the approach we propose in this thesis. Open-source tools are also limited in terms of computed performance metrics, and most of the identified open-source and commercial tools are lacking support for statistical tests validating the quality of collected data. Some research tools include statistical tests and data validation, as we do as part of the model-driven framework we propose in this thesis.

	Type	Test Definition Language	CSDL Integration	Metrics Computations
General Purpose Performance Testing Tools (Open-source)				
JMeter[JMeter, 1998]	Open-source	GUI, XML, Recording	✗	client-side, server-side
Faban[Faban, 2000]	Open-source	Java	✗	client-side, server-side
Gatling[Gatling, 2011]	Open-source	Scala, Recording	✓	client-side, server-side
PerfCake[PerfCake, 2016]	Open-source	XML	✗	client-side, server-side
Ascar Pilot[Li et al., 2016]	Research prototype	C++	✗	client-side, server-side
BenchExec[BenchExec, 2015]	Research prototype	XML	✗	client-side, server-side
DataMill[Petkovich et al., 2015]	Research prototype	Shell	✗	client-side, server-side
Taurus[Blazemeter, 2014]	Open-source	YAML	✓	client-side, server-side
Artillery[Shoreditch Ops, 2017]	Open-source	JavaScript, Recording	✓	client-side, server-side
Locust[Locust, 2020]	Open-source	Python, Recording	✗	client-side, server-side
The Grinder[Philip Aston, 2014]	Open-source	Java, Jython, or Clojur	✗	client-side, server-side
Tsung[Nicolas Niclausse, 2017]	Open-source	XML	✗	client-side, server-side
k6[Load Impact AB, 2019]	Open-source	JavaScript, Recording	CLI	client-side, server-side
ElasTest[ElasTest, 2016]	Open-source	Java, Recording	CLI, Plugins	client-side, server-side
Predator[PayU, 2020]	Open-source	GUI, JSON	✗	client-side, server-side
AutoPerf[Apte et al., 2017]	Research prototype	Ruby	✗	client-side, server-side
WS-TaaS[Yan et al., 2012]	Research prototype	GUI, XML	✗	client-side, server-side
Weevil[Wang, 2006]	Research prototype	Custom scripts	✗	client-side, server-side
Dynamic Spotter[Sopeco, 2014]	Research prototype	Declarative DSL (Custom DSL supporting Goals)	✗	client-side, server-side
PET[Brad Kemp, 2001]	Research prototype	Custom scripts	✗	client-side, server-side
General Purpose Performance Testing Tools (Commercial)				
HP LoadRunner[HP, 2016]	Commercial	GUI, C, Javascript, Java, Recording	CLI, Plugins	client-side, server-side
NeoLoad[NeoLoad, 2016]	Commercial	GUI, Modeling Languages (Partially Declarative), Javascript, Recording	CLI, Plugins	client-side, server-side
Akamai[Akamai, 2020]	Commercial	GUI, Recording	CLI, Plugins	client-side, server-side
Apica[Apica, 2016]	Commercial	GUI, Recording	CLI, Plugins	client-side, server-side
Dynatrace's Load[Dynatrace, 2016]	Commercial	GUI, Recording	✗	client-side, server-side
BlazeMeter[Blazemeter, 2016]	Commercial	GUI, XML, YAML	CLI, Plugins	client-side, server-side
WebLOAD[Radview, 2020]	Commercial	GUI, Recording	CLI, Plugins	client-side, server-side
LoadComplete[SmartBear, 2016]	Commercial	GUI, Recording	CLI, Plugins	client-side, server-side
IBM's Rational Performance[IBM, 2020]	Commercial	GUI, Java, Recording	CLI	client-side, server-side
LoadStorm[CustomerCentrix LLC, 2020]	Commercial	GUI, XML, Recording	CLI, Plugins	client-side, server-side
SilkPerformer[MicroFocus, 2020]	Commercial	GUI, C#, Recording	CLI, Plugins	client-side, server-side
RETiT[RETIT GmbH, 2020]	Commercial	GUI, Recording	CLI	client-side, server-side
BlackFire[Blackfire, 2014]	Commercial	GUI, YAML, Recording	CLI, Plugins	client-side, server-side
Middleware Oriented Performance Testing Tools				
SOABench[Bianculli et al., 2010a]	Research prototype	Declarative Scripts (Custom DSL)	✗	client-side
SOArMetrics[Li et al., 2009]	Research prototype	Declarative Scripts (Custom DSL)	✗	client-side, server-side
Cloud and/or Cloud-native Oriented Performance Testing Tools (Open-source and Commercial)				
Cloudstone[Sobel et al., 2008]	Research prototype	Shell	✗	client-side, server-side
Cloud Work Bench[Scheuner et al., 2014]	Research prototype	Declarative DSL (Ruby, Chef, Vagrant)	✗	client-side, server-side
Smart CloudBench[Chhetri et al., 2016]	Research prototype	GUI	✗	client-side, server-side
cbtool[Silva and Hines, 2016]	Open-source	Shell	✗	client-side, server-side
CloudPerf[Michael et al., 2017]	Research prototype	XML	✗	client-side, server-side
CloudTest[Podelko, 2016]	Commercial	Javascript	CLI	client-side, server-side
radon-ctt[RADON, 2020]	Research prototype	GUI, Tosca model	CLI	client-side, server-side
PerfKitBenchmarker[Google Cloud Platform, 2020]	Research prototype	Shell	✗	client-side, server-side

Table 3.1. Performance Testing Tools Overview

3.8 Concluding Remarks

In this chapter we present an overview of the state-of-the-art on performance engineering, focusing in particular on performance testing, one of the activities part of performance engineering practices. We then introduce the DevOps approach and the CSDL process and how performance testing is integrated with such approach and processes. We present the different research projects, research work, and commercial solutions and how they propose to integrate performance testing as part of CSDL and the challenges identified in doing so. We then present a brief overview of the main approaches and tools for the two main target systems of our work, namely RESTful Web services and BPMN 2.0 WfMSs. We offer an overview of the state-of-the-art in Web services and WfMSs performance testing automation, discussing work identified in academic and industrial literature. In the last part of the chapter, we introduce DPE and discuss its proposition towards simplifying performance engineering practices definition and increasing adoption of the same. We then particularly focus on DPE when applied to declarative performance testing, the objective of this thesis. We present different declarative DSLs available in the literature, allowing users to declaratively specify performance tests. The DSL proposed as part of this work is richer in terms of support for different types of performance tests, and enables goal-driven specification of tests and for complete control of the end-to-end test execution processes when paired with the model-driven framework we propose. We also present different state-of-the-art tools enabling performance tests execution and automation and highlight the ones offering part of the functionalities in a declarative way. Few tools offer solutions implementing DPE approaches. None of the tools offer an integrated approach for declarative performance test specification and automation in CSDL.

Part II

Declarative Performance Testing Automation

Chapter 4

Automation-oriented Performance Tests Catalog for DPE

In this chapter, we tackle the R.G. 1, and we present an automation-oriented performance test catalog, looking at the factors and contexts defining the different performance test types, and impacting the performance test automation. We provide an overview of the different performance tests types that have been defined both in academic and industry literature, we present their definition and we discuss different aspects useful when defining a DPE approach for defining and automating such tests, as well as how to select the different types of test in the context of DevOps. We refer in particular to the case of RESTful Web services and BPMN 2.0 WfMSs since they are the main SUT types on which we focus in the context of this thesis. Additionally, we mainly look at performance tests meant to measure and learn about SUT's performance, thus omitting some variants of discussed performance tests approaches designed for the system's performance optimization and fine-tuning.

4.1 Introduction and Overview

Different approaches of performance testing exist, from unit performance testing looking at the method and instruction-level performance, through end-to-end performance testing looking at the overall behavior of the software under different loads deployed in a production-like environment, up to live performance testing leveraging live traffic on user-facing deployed software for taking decisions about switching to the next release for the entire user base. As part of the effort in defining a DPE approach for performance test definition and execution, it is important to identify all different types of performance tests,

their goals, and the factors impacting their automation. A DPE approach abstract away complexity from the users, thus all the complexity of executing performance tests has to be identified, modeled, and removed from non-expert users' eyes, allowing expert users to still interact with all the aspects of defining performance tests. To do so, we need to characterize the goals and users' expectations of the different performance test types, their different configuration options and input parameters, the requirements towards the execution environment where the agent simulating the load are deployed, as well where the SUT is deployed. We also need to model the process to be followed to execute the different types of performance tests, as well as criteria of executions, checks to be performed during the execution, measurements to be collected and metrics to be computed before, during, and after the execution. Since the execution of different test types happens in different moments of the CSDL lifecycle, the SUT might not be completely ready, thus another important aspect is to identify how to deal with potentially missing software dependencies, e.g., by relying on mocks and simulation for some parts of the system. This is important also for some tests, for example, load tests, that have to be executed in isolation pointing to a specific Web service, isolating the boundaries of such service in a way that the collected measurements are not impacted by dependant services. Last but not least, for a successful automation-oriented performance test definition, it is important to identify the state and behavior of the SUT before, during, and after the execution to determine if the test is successful and to correctly compute the performance metrics. It is also important to define termination criteria to control the performance test duration, as well as quality gates to automatically decide whether a SUT reaches expected quality after the correct execution of the performance test has been assessed and validated.

4.2 Automation-oriented Performance Tests Definition

In this section, we present the contexts and factors we identified as defining the different types of performance tests available in the literature and impacting their automation process, and we provide a sample overview of performance test types alongside each factor. We then discuss all the factors to look at when reasoning about performance tests automation, and how they shape and impact different configurations and settings for the same type of performance test.

4.2.1 Contexts and Factors Defining Performance Tests Types

Focusing in particular on RESTful Web services and BPMN 2.0 WfMSs as SUT, we mainly consider two contexts defining performance tests, namely the SUT and the Client contexts. The SUT context looks at factors deriving from the technical point of view of software development, such as system design, system runtime, development process, and deployment environment. The Client context involves factors closer to the users (both humans or other systems) utilizing the system, such as their expectations, the device they use to access the system, and the way they expect to access such a system in terms of workload.

The SUT Context

The factors we identified as relevant for identifying the different kinds of performance tests to be defined, concerning the SUT context, are the system design, the system runtime, the followed development process, and the deployment environment.

System Design - The system design factors we identify as relevant for the scope of the catalog are related to the different levels of abstraction in looking at a software system and its realizing components and/or services. The lowest level of abstractions is the lines of code, the system's methods, and also classes when using object-oriented programming. At this level we see the definition of performance unit tests [Horký et al., 2015] employed to measure and capture performance variations at the lower possible unit and as soon as possible during the CSDL. The next level of abstraction in terms of the system design is the service when working with distributed systems. At this level, it is very common to define and execute performance tests such as smoke, load, and scalability [Neotys, 2018]. These tests can be executed targeting a single service in isolation, by relying on test doubles [Meszaros, 2007] to isolate the services from dependencies. The last level of abstraction for what concerns the system design is the system (or application) itself. Most of the tests executed at the service level are also executed at the application level, involving more than one service and looking at the overall behavior of the application. One type of test that is specifically executed at the application level, is the capacity or endurance test [Almeida et al., 2004; Molyneaux, 2014]. A capacity test is a test to determine how many users the SUT can handle without violating performance acceptance criteria or SLA.

System Runtime - The system runtime has a central role in defining the types of performance tests to execute and the metrics to report and analyze. The first characteristics we identify at this level is the programming language

the system is developed in, and its interpreter or language-specific virtual machine (e.g., the Java Virtual Machine (JVM)). It is common for interpreted languages or languages executed using a virtual machine, to have dedicated performance benchmark suites [Blackburn et al., 2006] executed to optimize the performance and the configuration of such infrastructure for the developed software and specific workloads. Dedicated performance benchmark suites are commonly used also when the application relies on a specific runtime to execute, for example in the case of Servlet Engines such as Apache Tomcat and the Internet Information Services Web server developed by Microsoft. Other important characteristics are the operating system and hardware runtime the application is deployed to, and if the hardware is physical, virtualized (e.g., using Hypervisors), emulated (e.g., using Virtual Machines) and the operating system is dedicated or shared (e.g., in the case of Container technologies). The fact the hardware is physical or virtualized, and the actual virtualization technology has a substantial impact on the process of executing performance tests because most often it undermines the stability and the reproducibility of tests' results and the reliability of the same as a base for making decisions about the system's deployment.

Deployment Environment - The environment the software is deployed to is another very important factor influencing the types of performance tests that are executed. Systems can be deployed on many different types of environments, with physical or virtualized system runtimes. Some deployment environments are mobile, personal computer, embedded systems, custom hardware, physical servers, private/public/hybrid Clouds. Each environment has its specific characteristics and requirements for performance test execution. Some of them are more challenging and complex than others, thus requiring specific types of performance tests or fine-tuned execution processes of the same. An example is the Cloud computing environments, in particular public and hybrid Clouds. Such environments usually offer infrastructure-level scalability and elasticity, thus scalability [Smith and Williams, 2001] and elasticity [Herbst et al., 2013] performance testing of the application is often performed to assess the ability of the application to rely efficiently on such infrastructure features. Other tests usually executed in this context, specifically because such environments are selected for application requiring performance flexibility, are stress [Molyneaux, 2014] and spike performance tests [Guru99, 2014]. Some computing environment, e.g., the mobile one, are characterized by unstable and unpredictable network performance and conditions, thus requiring specific kinds of performance tests targeting the network-level to assess the conditions are sufficient for operating the application, such as the throttle performance

tests[1].

Development Process - Last but not least, the deployment process certainly affects the kind of tests that are executed, and even more the process that is followed to executed such tests. Many different development processes exist, that are characterized, among other things, by the sequence in which software development and release tasks are performed, how many iterations are undertaken for releasing the first software version and its successive releases, the time in between such iterations, and the way new releases are exposed to end-users (e.g., with some downtime, multiple versions in parallel, etc.). Examples of such development processes are Waterfall, Agile, Spiral, DevOps, and Continuous. Definition and execution of performance tests are mostly affected by the velocity of the process and the number of iterations of the same, as well as the way new releases are exposed to end-users. A type of test that is always executed, independently of the process, when the system is considered ready for release is the capacity or endurance test [Molyneaux, 2014]. There are different forms of endurance tests, such as soak or stability test [Molyneaux, 2014], and their objective is to assess the system can perform as expected over an extended period of time. In processes with multiple iterations, such as the Agile, DevOps, and Continuous ones it is common to perform exploratory testing [Guru99, 2014] in the initial phases of the development, and in each iteration introducing unknown and breaking changes. When a system's feature reaches good enough stability it is common to introduce acceptance[2] and regression [Liu, 2011] performance testing to ensure future changes do not break the performance of more stable features. In development processes such as the DevOps and the Continuous one, it is also common to expose new releases to the end-users in an incremental manner, thus it is common to execute live-traffic or canary performance testing [Schermann et al., 2018] to drive release processes such as canary release by mirroring real end-users traffic. In the same processes, it is common practice also to execute chaos performance testing[3], to ensure self-healing characteristics and the ability to sustain known failures that applications developed and deployed following such processes are expected to implement.

[1] https://abstracta.us/blog/performance-testing/types-performance-tests/, last visited on February 7, 2021

[2] https://www.istqb.org/, last visited on February 7, 2021

[3] https://netflixtechblog.com/the-netflix-simian-army-16e57fbab116, last visited on February 7, 2021

The Client Context

The factors we identified as relevant for identifying the different kinds of performance tests to be executed, concerning the client context, are the client's expectations related to performance, the device utilized to access the system, and the expected workload to be generated in accessing the system.

Performance Expectations - Different clients have different performance expectations, also according to the kind of system they are connecting to. In the context of the thesis, we mainly see requirements towards more or less strict SLAs e.g. in terms of the response time of the service, the availability requirements of the service, and related to the criticality of the service. Guaranteeing more strict SLAs usually requires more extensive performance tests, to verify the systems guarantees such SLAs also when sub-optimal conditions are experienced. In this context performance tests such as breakpoint[4] and failover or recovery[5] are commonly executed. Systems that are expected to guarantee near 100% availability are usually assessed relying on specific kind of tests such as the resiliency or reliability[6] performance test checking for the ability of the system to smoothly recover from failures and unexpected behavior without affecting the availability and the performance of the same. The same kinds of tests are applied and very relevant also for highly critical systems, such as systems deployed in healthcare and aeronautics. For such systems, it is also common to execute extensive configuration tests [Guru99, 2014] to fine-tune and assess the system's configuration guaranteeing the required performance and stability of the system.

Utilized Device - The device utilized to access the system has also an important role in determining the kind of tests that needs to be executed. We already covered most of the specific situations when presenting the tests according to the development environment. The device utilized to access the system adds additional requirements towards the collected measurement data and the environment in which those are collected. For example, in the case of an application accessed in a browser, it is important to execute performance tests also when the application runs in the actual browser the users' are expected to use. In the case of a mobile device it is common to execute specific kinds of performance test called snapshot-load performance tests[7], to check for

[4] https://www.perfmatrix.com/application-break-point-test/, last visited on February 7, 2021

[5] https://qa-platforms.com/what-is-failover-testing/, last visited on February 7, 2021

[6] https://www.ibm.com/developerworks/websphere/techjournal/1407_col_nasser/1407_col_nasser.html, last visited on February 7, 2021

[7] https://jestjs.io/docs/en/snapshot-testing, last visited on February 7, 2021

the correct representation of the application on the user's side, given specific performance constraints and conditions.

Generated Workload - The workloads the client expects to generate when accessing the system, also impose requirements towards the tests that are executed. Workload shapes can be of different forms, in some cases are known and can be defined in advance, in others are unpredictable. Examples of workloads are one-shot events, repeating events with a given pattern, events happening once every a given amount of time, clients connecting with short or long sessions, push-based workloads, pull-based workloads (e.g., in the case of passive publish/subscribe), fire-and-forget events, streaming events. Tests that are usually defined for specific workloads are the peak-load performance test [Guru99, 2014], for testing the expected heaviest demand that would be generated at peak client times, as well as tests such as spike and stress performance tests already mentioned above in the text. Another test that is executed when the application is expected to generate and or collect a considerable amount of data over time, is the volume or flood performance test [Guru99, 2014]. Volume testing is performed to assess the system performance by increasing the volume of data stored in the datastores the system relies on.

Independently on the context and type of executed test, is good practice to always execute a baseline performance test [Molyneaux, 2014] to measure and report baseline performance of the system that is expected to be guaranteed, for example for identifying reference points for other tests and the system's performance plateau.

4.2.2 Factors Impacting Performance Tests Automation

Different factors define the requirements performance tests automation processes have to comply with. Some of the factors depend on the above-mentioned contexts, as already presented, some of the others are very performance test specific, and depend on the expectations about performance tests' results. The main factors we identified are assumptions on the SUT maturity, expectations on the execution environment conditions, input parameters, required execution process, and checks to be performed on the SUT, preliminary performance tests to be already executed, measurements to be collected and metrics to be calculated. The SUT maturity impacts the automation process of a performance test in the sense that it sets specific requirements towards the same. A SUT in its early stages requires exploratory tests to learn about its performance and to set requirements to satisfy its missing dependencies that might not be ready to be deployed. According to the SUT and the performance test types,

different requirements are set towards the execution environment conditions. A SUT deployed in certain environments, for example on public Cloud, usually have expectations in terms of execution environment conditions that are more relaxed compared to systems deployed on dedicated hardware. This impacts the way performance test automation is carried out because, to obtain reliable metrics on environments affected by more variance, a different process needs to be followed [Arif et al., 2018; Bachiega et al., 2018] for the same test type. For specific kinds of tests, for example, regression testing is important to select very similar execution conditions for each test repetition otherwise the risk is to increase false positives in performance regression identifications. This is another example of a requirement imposed on the execution environment conditions.

Another important aspect to consider when defining test types is the input parameters to the test. Different test types have different requirements for input parameters. Consider for example tests requiring certain data to be available in the SUT before test execution can start. This requires taking care of data preparation and setup before the actual test execution can start. Likewise, certain input parameters could also be generated during the test execution itself. One example is data generated across different requests to services, to be used to subsequent requests to other services. This impacts the automation process definition, which needs to take into account this requirement so that this kind of interaction can be automated.

When looking at the automation process itself, it needs to be driven in such a way that the process itself is safe and guarantees the required checks and verifications according to the test types. A common check needing to be performed while automating performance tests is the verification of the SUT reaching the steady-state. This is important for some kinds of performance tests, for example, to decide the moment from which to start measuring the performance metrics of interest.

A very important factor impacting the performance test automation process is the interdependencies across different test types. A solid automation process must consider interdependencies across tests, and validate dependent tests are being executed before scheduling the current test. This is important also for deciding the priority of test execution when multiple tests are waiting for execution due to competing access to shared test resources. Executing a complex test, such as an endurance test, is for example not advisable if the system has not been tested with more simple and lightweight performance tests such as smoke tests. It is of course responsibility of the performance test engineer to decide to execute different tests, but it is due to the automation process taking into account the dependencies across tests and warn the engineer relying on

the automation process.

Last but not least, another important element impacting the performance test automation process are the metrics to be computed. Different metrics pose different requirements to the process carried out to execute the performance tests and impose requirements on tools to be put in place to collect measurements the metrics depend on to be computed. Another factor related to metric impacting the automation process is the precision we express for the metric. Consider for example the case we want to measure CPU utilization, and we expect this measurement to experience a relatively low error in the measurement to increase the reliability of the obtained data. To guarantee the specified error to be respected the automation process, also according to additional conditions (e.g., the deployment environment), needs to carry out the test for a more or less longer duration as well as needs to schedule a specific number of repetition of the test to provide with reliable measurements to the user of the automation process.

4.3 Automation-oriented Performance Tests Catalog

In this section, we introduce our automation-oriented performance tests catalog following a catalog representation template proposed by us to collect and present all the relevant data to be used to model the automation requirements. The catalog is not meant to be exhaustive, but to present in an automation-oriented definition all the test types that are common when dealing with applications and the processes in the scope of this work.

We structure the template we use to describe our catalog to comply with the data deemed to be useful for a performance test taxonomy[8], and we describe a set of performance test types referring to them by their names. The objective of the catalog is to describe the different performance tests concerning their automation process requirements and the relationships existing between them.

4.3.1 The Performance Tests Catalog Template

To facilitate the access to the data needed for a DPE approach for test definition and automation, we define a precise set of data to be provided to present each type of performance test we identified.

[8] https://www.if4it.com/SYNTHESIZED/GLOSSARY/S/Software_Performance_Test_Taxonomy.html, last visited on February 7, 2021

Each performance test in the catalog is described by the structure presented in Listing 4.1.

- Performance Test Name
  - Definition
  - Objectives
  - Expected Input
  - Expected SUT Maturity
  - Expected Preliminary Performance Tests
  - Execution Environment Requirements
    - Before the Test
    - When Starting the Test
    - During the Test
    - Before Concluding the Test
    - After the Test
  - Execution Process
    - Before the Test
    - When Starting the Test
    - During the Test
      - Suspension Criteria
      - Exit Criteria
    - Before Concluding the Test
    - After the Test
  - Test Deliverables
    - Metrics
    - Quality Results
    - Reports

Listing 4.1. The Performance Tests Catalog Template

The *Performance Test Name* is a unique identifier of the test and identifies the test in the catalog. Each test in the catalog is accompanied by a *Definition*, in most cases obtained from the literature about performance testing, and in some cases extended to better clarify the importance of the test in the context of DPE. The definition enables us to clearly state the test *Objectives*, a very

important element to automate the test execution relying on DPE techniques. We then specify the *Expected Input*, required for a correct test instantiation and execution. The expected input enables a thorough assessment of the feasibility of test execution concerning expected input data to be available. We include in the test input both the test data and the issued workload. The *Expected SUT Maturity* then defines the requirements the test set towards the SUT in terms of how far it is in the development process of the same. This element is very important to ensure the test can be executed, thus automated, according to the state of the SUT. To complete the data useful for identifying if a test can be executed, we report the *Expected Preliminary Performance Tests*. This data allows the automation process to optionally check whether dependant tests have been already executed or not, to optimize performance tests execution, well-known to be time and resource consuming.
Execution Environment Requirements - The execution environment requirements look at the requirements of the execution environment necessary to guarantee a successful and correct performance test automation. The execution environment includes the environment in which the SUT is deployed, as well as the environment in which the performance test automation is deployed. Concerning the environment it is important to look at:

a) *Before the Test*: defines the expected conditions the test execution environment have to demonstrate before the performance test can be scheduled for test execution;

b) *When Starting the Test*: defines the expected conditions the test execution environment have to demonstrate at the moment in which the simulated load is issued against the SUT;

c) *During the Test*: defines the expected conditions the test execution environment have to demonstrate and maintain during the execution of the performance test;

d) *Before Concluding the Test*: defines the expected conditions the test execution environment have to demonstrate before concluding the performance test execution;

e) *After the Test*: defines the expected conditions the test execution environment has to demonstrate after the performance test is considered complete.

Execution Process - The execution process refers to actions to be considered to automate the test in different moments of the test execution lifecycle. Automating a performance test includes many different actions alongside the actual performance test execution in terms of workload issued to the SUT. Concerning the execution process, it is important to look at the following moments in the lifecycle:

a) *Before the Test*: reports the steps to be executed before the execution of the performance test;

b) *When Starting the Test*: reports the steps to be executed when the performance test starts;

c) *During the Test*: reports the steps to be executed during the execution of the performance test, while the workload is issued;

    a) *Suspension Criteria*: reports the steps to be executed *during the test* execution to determine if the performance test has to be suspended, to avoid waste of resources and/or never-ending tests;

    b) *Exit Criteria*: reports the steps to be executed *during the test* execution to determine when the performance test can be concluded, whether it is for successful completion or in case of non-recoverable errors.

d) *Before Concluding the Test*: reports the steps to be executed right before proceeding with the conclusion of the performance test;

e) *After the Test*: reports the steps to be executed after the performance test has been concluded, and no workload is issued to the SUT.

Test Deliverables - The test deliverables describe the expected artifacts and outcomes to be produced by the test, during, and/or at the end of the test execution. The deliverables to be produced and their requirements are fundamental, thus is important to know them for successful test automation, because they impact the way the automation is performed. Concerning the test deliverables, it is important to express the following data:

a) *Metrics*: specifies the expected metrics to be computed, and precision criteria expressed on the same;

b) *Quality Results*: specifies the criteria for defining whether the SUT successfully or not satisfies the expected quality criteria;

c) *Reports*: specifies the expected report to be generated, indicating the information about the performance test execution and performance test outcome.

The proposed template guarantees a complete definition of performance tests in terms of their requirements, and empower us to propose a DPE approach for performance test automation, implementing all the requirements expressed in the template. Must be noted that not all performance tests define all the different elements of the template. For some tests, some of the elements are not needed.

4.3.2 The Performance Tests Catalog

What follows is the proposed automation-oriented performance tests catalog for DPE. The performance tests are proposed in no particular order because an order is not possible to be enforced. We favor mentioning first performance tests on which other tests depend on and tests applied on the system starting from its initial stage of existence up to the system being deployed in production. When presenting the tests using the proposed catalog template, we deep-down in details referring to the two main contexts we presented, that helped us to identify the tests part of the catalog, every time such contexts impact the test definition and its automation. Such contexts are the client context, referring to the performance expectation, utilized device and generated workload, and the SUT context, referring to the system design, the system runtime, the deployment environment, and the development process.
All the performance tests share some aspects referred to the *Execution Environment Requirements*, the *Execution Process*, and the *Test Deliverables* when they are defined for being implemented following a DPE approach. We report in the following the shared aspects, and we omit to repeat them for the individual tests. For the individual tests, we report additional details enriching some of the shared aspects if deemed useful.

Execution Environment Requirements: ensure the execution environment is as close as possible to the SUT production deployment environment;

Before the Test: ensure that *a*) the execution environment is ready for the execution, the resources required for executing the test are available, and no other executions are in place in the same environment if that has to be guaranteed and *b*) the SUT is correctly deployed in the execution environment, if that is required for the test being executed;

When Starting the Test: ensure that the tools needed for collecting the performance data useful to compute the specified metrics are correctly deployed and ready to be started;

During the Test: ensure that resources are freed up as expected, to avoid unexpected noise on the test results;

Before Concluding the Test: ensure that the tools needed for collecting the performance are successfully undeployed;

After the Test: ensure that *a*) the SUT is correctly undeployed from the execution environment, if that is required for the test being executed and *b*) the execution environment is left clean and as it was before the test execution;.

Execution Process: no specific ones;

Before the Test: ensure that the SUT is ready for receiving the workload;

When Starting the Test: ensure that the tools needed for collecting the performance data are started, or schedule the collection start for a deferred time according to the state of the SUT (e.g., when the steady-state is reached);

During the Test: ensure that the tools needed for collecting the performance data are correctly collecting the expected data, and the data can be considered correct and valid for the scope of the test;

Suspension Criteria: ensure that *a*) time-based suspension criteria are in place, to control the execution environment resource allocation in such a way different tests can be scheduled and executed; *b*) monitoring-based suspension criteria are in place to identify flaky tests or tests having a high probability of failure due to the behavior demonstrated during the execution of the same; *c*) safety suspension criteria are in place, to avoid the test execution could monopolize all the available resources in the execution environment, reaches deadlock, livelock, starvation, or goes into an endless loop; *d*) preemption-based suspension criteria are in place, to be able to schedule unforeseen performance tests deemed to be of a higher priority compared to the ones currently in execution;

Exit Criteria: ensure that criteria to evaluate the exit state of the test execution are defined and consistent such that is possible to determine if the performance test is successful, failing, or undecided according to the specified criteria. The criteria often refer to metrics values and relative descriptive statistics.

Before Concluding the Test: ensure that the tools needed for collecting the performance data are successfully stopped and performance data are successfully collected;

After the Test: ensure that the test result is assessed and a quality result is provided.

Test Deliverables:

Metrics: descriptive statistics of metrics represented by many data points, usually measured over time, are usually reported. Ensure to report at least: quartiles [Montgomery and Runger, 2013], 90th, 95th, 99th percentiles [Montgomery and Runger, 2013], minimum and maximum value, as well as the average and the standard deviation;

Quality Results: Ensure that the reported quality results are consistent with the test objective, and clearly state how reliable the result can be considered according to the test objective and specified workload, computed metrics, the execution environment state, and the execution process outcome;

Reports: the report must report all the data useful to ensure the quality of the test is the expected one. Important information to ensure the defined test has been executed as specified are: Little Law [Little, 1961] and think time deviation [Faban, 2000]. If the Little Law is failing, it is important to report details about the clients issuing the workload, usually in terms of capacity [Molyneaux, 2014], and system resource statistics, such as *CPU*, *Random Access Memory (RAM)*, *Disk Storage (DISK)*, *Network (NET)*. For what concerns the state of the SUT, it is important to report the percentage of errors incurred during the test execution. If errors are present, it is important to include in the report details about such errors.

In the following, we list all the tests part of the proposed catalog. The complete list is comprised of the following test types: Baseline Performance Test, Unit Performance Test, Smoke Test, Performance Regression Test, Sanity Test, Load Test, Scalability Test, Elasticity Test, Stress Test, Peak Load Test, Spike Test, Throttle Test, Soak or Stability Test, Exploratory Test, Configuration Test, Benchmark Performance Test, Acceptance Test, Capacity or Endurance Test, Live-traffic or Canary Performance Test, Chaos Test, Breakpoints Performance Test, Failover or Recovery Test, Resiliency or Reliability Test, Snapshot-load Test, and Volume or Flood Test.

1. Baseline Performance Test

Definition: a baseline test is defined as a reference point for the SUT's performance measured according to a well-defined input set, stable SUT configuration, and given test execution environment, especially for what concerns the SUT deployment environment. A baseline test is set to provide a "best case" representation of the SUT' performance. A baseline test is updated as the SUT is developed and a new reference point has to be computed if the reference input or SUT configuration change, as well as when the test execution environment changes [Molyneaux, 2014].

Objectives: the main objectives of executing baseline tests are: *a*) to measure a reference point for the SUT's performance to be referred to when executing other tests and *b*) to validate the SUT is ready for more complex performance tests. If a SUT is not able to sustain a baseline test, is improbable can sustain performance tests issuing more workload, especially the performance test for which the baseline is measured for.

Expected Input: the expected input depends on the test for which the baseline test is executed. Usually, the input for the baseline test is less complex and more lightweight compared to the test type for which the baseline is measured for.

Expected SUT Maturity: the baseline test has no particular requirements towards the maturity of the SUT because such test is executed as a reference point for many other performance tests setting specific requirements on the SUT that the baseline test has to follow according to the performance test for which the baseline test is executed for.

Execution Environment Requirements: *Before the Test*: ensure the environment is as close as possible to the one that is going to be utilized when executing the test for which the baseline is measured for.

Test Deliverables: *Metrics*: the metrics depend on the test for which the baseline is measured for and must be consistent in terms of required precision.

2. Unit Performance Test

Definition: a unit performance test is defined as a test executed at a specific unit of the source code of the SUT, usually at method level [Johnson et al., 2007].

Objectives: the main objectives of executing unit performance testing are: *a*) to provide developers with early feedback on the performance behavior of the built software, as soon as some units of the same are developed and *b*) to enable low-level performance analysis, isolating the performance of specific units during the execution of the SUT.

Expected Input: the expected input is modeled after defining the input of the unit for functional testing. The same principles apply also for performance testing, to cover the behavior of the SUT across the entire domain of expected or unexpected input data and workload.

Expected SUT Maturity: it is sufficient some of the units are developed. For unit performance testing one does not need the entire SUT to be ready. The higher the completeness of the SUT's development, the more reliable the results represent actual SUT performance.

Execution Process: no specific ones;

During the Test: ensure the number of repetitions of each test case is sufficient to guarantee the reliability of measured performance [Bulej et al., 2014]. Looking at units the variability in performance test results of subsequent execution of the same test case could demonstrate a wide variance, thus it is important to dynamically adjust the number of iterations according to the obtained results;

Test Deliverables:

Metrics: it is important to measure the execution time of the unit. Other metrics, e.g., referred to resource utilization are not advisable to measure at the unit level because they are unreliable [Bulej et al., 2014];

Reports: present low-level unit execution data about the performance of the unit execution, useful to investigate the performance behavior. If the SUT runs on virtualized environments, present also the information about the performance of the underlying stack.

3. Smoke Test

Definition: a smoke test performs a lightweight test to ensure critical SUT functionalities work as expected [Guru99, 2014].

Objectives: the purpose is to identify broken builds of the SUT and do not proceed to more complex tests.

Expected Input: a subset of the input domain expected by the SUT, usually comprising of input data the SUT is designed to accept. The issued workload is generally characterized by a low intensity of issued work and a workload function the SUT is expected to handle without foreseeable issues.

Expected SUT Maturity: the SUT is mature enough to be deployed in the expected execution environment.

Expected Preliminary Performance Tests: *Baseline Test*. A baseline test is usually less resource-demanding than the smoke test, thus is expected to be executed successfully before scheduling the execution of a smoke test.

4. Performance Regression Test

Definition: performance regression testing involves the re-execution of previous tests following a set of changes, to identify any new faults that may have been introduced, directly or indirectly, as part of the code modification [Liu, 2011].

Objectives: capture SUT performance behavior over different builds, for critical SUT functionalities and main SUT's utilization scenarios.

Expected Input: the most common input subset for the selected functionalities and scenarios, to cover as close as possible expected or known users' utilization. In terms of the workload, regression tests usually cover the foreseeable or known real users' workload functions, favoring the execution of low-intensity functions first to catch issues as soon as possible and block the execution of the regression test suite.

Expected SUT Maturity: it depends on the level in the system design where the regression test it is applied to. Performance regression tests can be applied from the unit level, all the way up to the system level.

Expected Preliminary Performance Tests: *Smoke Test*. A smoke test is expected to be executed before regression testing because it is important to first check new or updated functionalities are working as expected before comparing the performance across different builds.

Execution Environment Requirements: ensure that the execution environment is as close as possible across multiple iterations of the regression test

suite executed against different builds. The noise introduced by the execution of the test in the environment has to be verified and is important to be similar across repetitions. In some deployment environments, such as on public Clouds, this is not always possible due to the characteristics of the environment itself. In such cases, it is advisable to execute performance regression tests in a dedicated environment guaranteeing a deterministic variance on the results;

Execution Process: agile and iterative development processes impacts the execution of the performance regression test suite, due to the time usually needed to execute such a suite and the velocity of the iterations. In such cases is important to identify when in the development flow to execute such suite, for example executing it only when merging changes related to a feature to the mainstream development code and before merging development code to production code. It is also common to split the suite such that most critical tests are executed more often, while the entire suite is executed less frequently and only at critical moments in the SUT's development lifecycle [Biswas et al., 2011].

Before the Test: ensure the reference test executions are available for the regression criteria to be verified. For this kind of test is important also to have available the knowledge about the state of the execution environment of reference executions;

Suspension Criteria: suspend the regression test suite a soon as one test does not fulfill the regression criteria. It is important to save execution time, and although one might need multiple iterations to have a complete set of results of performance regression tests, is better to suspend the execution as soon as possible due to the difficulty in calculating cross-impacts of changes the SUT has to integrate to fix the first discovered regression issues;

Test Deliverables:

Metrics: define a small subset of performance metrics to be monitored for regression, in all the areas of focus for the SUT's performance. It is common to define the response time as a performance regression metric to be monitored [Jiang and Hassan, 2015]. For SUTs deployed on environments with resources constrains, such as mobile or edge devices, it is common to monitor resource utilization regressions as well;

Quality Results: the result has to explicitly report whether the SUT has regressions or not. It is important to express the regression criteria such that minimum and a target value for each given monitored metric is provided. This allows to check for regression against both the minimum and the target value, providing a better understanding of how close the system is to reaching performance requirements or regressing from those;

Reports: it is important to provide the answer to the test objective clearly and concisely. Must be clear whether regression is present or not, and the reasons leading to the regression.

5. Sanity Test

Definition: a sanity test performs a lightweight test to ensure updated and new functionalities work as expected [Guru99, 2014].

Objectives: identify the new or modified features that are far from complying with the defined performance criteria, before the code is merged to the main development branch. It compares to smoke testing in the sense that for both of them the test is lightweight compared to other test types and are not exhaustive in testing the performance behavior of the SUT;

Expected Input: as per the smoke test;

Expected SUT Maturity: as per the smoke test;

Expected Preliminary Performance Tests: *Regression Test*, and *Smoke Test*. The SUT has to be tested for regressions for a considerable amount of builds and has to be stable in terms of identified regressions. At this point, one can start to introduce the sanity test as part of the tests executed against the SUT, before running the entire regression tests suite. For this to be successful, smoke tests have to be always executed to guarantee critical functionalities of the system keep working as expected.

6. Load Test

Definition: "load testing is the process of assessing system behavior under load to detect problems due to one or both of the following reasons: (1) functional-related problems (i.e., functional bugs that appear only under load), and (2) non-functional problems (i.e., violations in non-functional quality-related requirements under load)". [Jiang and Hassan, 2015]

Objectives: the "[...] application is loaded up to the target concurrency but usually no further. The aim is to meet performance targets for availability, concurrency or throughput, and response time [...]". [Molyneaux, 2014] The load test focuses on the load issued to the SUT and the SUT ability to handle such load.

Expected Input: the expected input is designed to be as close to the real one as possible, being the load test the closest approximation of actual application use by real users. The workload is expected to be modeled very closely with the real users' behavior and interaction with the SUT. Whether actual real workload is available, load test inputs should be modeled starting from that. [Molyneaux, 2014]

Expected SUT Maturity: being the load test meant to represent real usage of the SUT, it requires the SUT to be mature enough to be deployed in the expected execution environment. It is not necessary the SUT is complete in terms of functionalities to start performing load tests. One can start performing load testing at the prototyping stage already.

Expected Preliminary Performance Tests: *Baseline Test*, *Smoke Test*, and *Sanity Test*. A baseline test is required to have a baseline reference for the load test, to be used along with the defined SLA. The smoke and the sanity tests are important to ensure the SUT features are working as expected before proceeding with the load test, which issues a much higher load.

Execution Environment Requirements: the load test is very resource-demanding both for real user load simulation and for the resource needed by the SUT to cope with such load. It is important to carefully size the resource needed so that the load test can be successful and not affected by unexpected issues due to limited resources. It is important the deployment environment and the SUT's deployment configurations are as close as possible to production.

Execution Process: no specific ones;

When Starting the Test: for SUTs for which a steady state is expected to be reached, is important to check the SUT reached the steady-state before starting to monitor and capture its performance;

During the Test: no specific ones;

Suspension Criteria: strict suspension criteria have to be in place, to avoid wasting resources, even stricter than criteria always in place independently of the test type. One important suspension criteria for the load test is one applied to steady-state detection. If the SUT is not able to reach the steady-state within the expected amount of time, it is advisable to suspend it. It is advisable to suspend the test execution also when the system is not able to sustain the steady-state and exhibits divergences from it during the test execution. In the steady-state, the average resource utilization must be constant within an acceptable range and linear to the issued load. The average response time and the average memory utilization should be constant as well, and free from trends of any kind [Bondi, 2014];

After the Test: given the sustained amount of load issued for the execution of a load test, it is important to ensure the execution environment correctly and safely returns to the state it was before the test execution and all the resources are freed up correctly.

Test Deliverables:

Metrics: the metrics of interest for a load test are usually referring to the defined SLAs and look at response time, throughput, and resource utilization [Barna et al., 2011];

Quality Results: the quality results report whether the SUT complies or not with the defined SLAs. In case SLAs are not defined, e.g. because the system is in a too early stage, the evaluation is based on setting a reference baseline for the load test and ensuring that at time t the SUT performance is at least the same as it was at the time $t-1$ for the tested functionalities [Huebner et al., 2000];

Reports: the report has to indicate the metrics of interest and all the metrics relevant to be reported when a system is under sustained load. The reported metrics depends on the deployment environment, but always comprises resource utilization in terms of *CPU*, *RAM*, *DISK*, *NET*, but also *Operating System (OS) threads behavior* descriptive statistics. For SUT executed on virtualized runtime, such as Java programs executed on a JVM, it is also important to collect runtime-specific metrics and performance behavior.

7. Scalability Test

Definition: a scalability test is designed to measure the "ability of a system to continue to meet its response time or throughput objectives as the demand for the software functions increases" [Smith and Williams, 2001]. It also ensures the SUT efficiently uses resources, and correctly scales down when the load diminishes, guaranteeing the response time throughput objectives are still met and the resources utilized are reduced accordingly.

Objectives: ensure the SUT reaches scalability criteria as defined in the test requirements. The criteria usually cover different aspects of scalability, namely: load scalability, space scalability, space-time scalability, and structural scalability [Bondi, 2014];

Expected Input: the expected input has to be modeled to represent the foreseen scalability expectations of the SUT. The expectations must be modeled in terms of the input data, as well as the workload function. More data are available, more the modeled function can be expected to better represent the actual load the system is going to experience in real-life;

Expected SUT Maturity: the SUT has to reach a maturity for which it makes sense to perform scalability testing. Usually, after the SUT reached the maturity for which load tests are executed, after some iterations it is also ready for scalability testing of the same functionalities;

Expected Preliminary Performance Tests: *Baseline Test*, *Smoke Test*, *Sanity Test*, and *Load Test*. A baseline test is required to set a baseline for the scalability assessment. Smoke and sanity test are important to ensure the functionalities tested for scalability are correctly working and demonstrating expected performance behavior. A load test is fundamental to guarantee the system is correctly handling load level expected to be experienced in production deployment, before testing the same functionalities for higher load levels;

Execution Environment Requirements: as per the load test;

Execution Process: no specific ones;

When Starting the Test: ensure the SUT is in a stable state, potentially a steady-state, before starting to issue load to test the scalability. This is important to ensure one can measure how the SUT reacts to higher or lower load, starting from a controlled state;

During the Test: no specific ones;

Suspension Criteria: it is important to constantly monitor the behavior of the system in scaling-up and scaling-down. If the SUT does not sustain changing load, one might decide to suspend the test to avoid using many resources and a considerable amount of time for a test execution that is likely to be repeated;

After the Test: as per the load test.

Test Deliverables: no specific ones;

Metrics: the metrics of interest are looking at different aspects of scalability, and usually covers: response-time, throughput, and resources utilization;

Quality Results: the quality result indicates whether or not the SUT scales according to the scaling load;

Reports: the report has to detail all the relevant aspects useful to evaluate how well the SUT scales with load, similar to the ones reported for the load test. In some deployment environments, services to control and handle scalability are in place, such for example on Cloud infrastructures or Container orchestrators. In such cases, it is important to report also metrics coming from such a piece of software, to provide a complete picture of how the system and the underlying infrastructure reacted to scaling load. In some cases, one can also report extrapolation built over the collected performance data and metrics, to provide a forecast on what can be expected over time, or for different scalability requirements in place.

8. Elasticity Test

Definition: an elasticity test measures the "degree to which a system can adapt to workload changes by provisioning and de-provisioning resources in an autonomic manner, such that at each point in time the available resources match the current demand as closely as possible." [Herbst et al., 2013].

Objectives: ensure the SUT matches elasticity criteria on the given deployment platform. Elasticity criteria are usually defined in terms of speed and precision to adjust to changing conditions.

Expected Input: as per the scalability test. Additionally, criterion based on which the amount of provisioned resources is considered to match the

actual current demand needed to satisfy the system's given performance requirements have to be defined;

Expected SUT Maturity: as per the scalability test, but additionally expecting autoscaling mechanism are in place;

Expected Preliminary Performance Tests: as per the scalability test, and including the *Scalability Test* as well. Testing the scalability of the system before testing for its elasticity ensures the system is ready for handling varying workload conditions.

Execution Environment Requirements: as per the load test, with additional elasticity mechanisms in place;

Execution Process: no specific ones;

Before the Test: ensure no autoscaling rules are triggered;

When Starting the Test: as per the scalability test. Additionally, it is important to record the starting deployment condition of the system, in terms of resource allocation and services instances;

During the Test: no specific ones;

Suspension Criteria: as per the scalability test, but applied to elasticity criteria;

After the Test: as per the load test.

Test Deliverables: no specific ones;

Metrics: the metrics of interest are looking at different aspects of elasticity, and usually covers: accuracy, timeshare, jitter, number of Service Level Objectives (SLO) violations, mean time to quality repair [Herbst et al., 2015];

Quality Results: the quality result indicates whether or not the SUT elastically adapts to the changing load;

Reports: as per the scalability test but applied to elasticity criteria.

9. Stress Test

Definition: a stress test is designed to determine the upper limits of a system's performance in a given configuration [Molyneaux, 2014].

Objectives: overload the SUT to the point it starts to fail or the supporting infrastructure starts to fail, and check the ability of the SUT to return to normal operation when the load diminishes. It is important to know when the system starts to fail, to better plan for capacity.

Expected Input: as per the load test, but with a higher load;

Expected SUT Maturity: as per the load test;

Expected Preliminary Performance Tests: as per the elasticity test. If elasticity testing can not be performed on the SUT, then as per the scalability test;

Execution Environment Requirements: as per the scalability or elasticity test;

Execution Process: no specific ones;

When Starting the Test: as per the scalability test;

During the Test: no specific ones;

Suspension Criteria: as per the scalability test with detailed detection of incurring failures during the stress test;

After the Test: as per the load test.

Test Deliverables: no specific ones;

Metrics: the metrics of interest are defined to characterize possible failure scenarios of the SUT. Metrics to be computed are: pages per second, hit time, throughput, time to the first byte, response time, number of failures [Guru99, 2014], as well as resource metrics such as RAM, CPU, DISK, NET [Molyneaux, 2014].

Quality Results: the quality result indicates the main failure points of the SUT and whether the system displays appropriate error messages while under stress;

Reports: the report indicates all the failing situations of the SUT and a wide spectrum of metrics representing the SUT performance both from the client-side and the server-side.

10. Peak Load Test

Definition: a peak load test is designed to place a load on a SUT for a short period of time to emulate the highest demand that would be generated at peak user times [Guru99, 2014].

Objectives: ensure the SUT can handle a sudden increase in the number of requests, defined according to known load situations of the SUT.

Expected Input: as per the scalability test, but applied to know peak load situation;

Expected SUT Maturity: as per the scalability test;

Expected Preliminary Performance Tests: as per the elasticity test. If elasticity testing can not be performed on the SUT, then as per the scalability test. Additionally, also *Stress Test* has to be performed preliminary, to ensure the system can handle variations in the issued workload;

Execution Environment Requirements: as per the load test, with additional consideration for the resources needed to handle the defined peak load. When sufficient resources are not available to handle the peak, one may consider reducing the resources allocated to the SUT and consequently, the peak load, to simulate the SUT's management of peak conditions[9];

Execution Process: no specific ones;

When Starting the Test: as per the scalability test;

During the Test: no specific ones;

Suspension Criteria: as per the stress test;

After the Test: as per the load test.

Test Deliverables: no specific ones;

Metrics: as per the stress test;

Quality Results: the quality results report the behavior of the SUT at peak load;

Reports: the report indicates all the successful/failing behavior of the SUT and a wide spectrum of metrics representing the SUT performance both from the client-side and the server-side.

[9] https://www.microfocus.com/documentation/silk-test/200/en/silktestclassic-help-en/STCLASSIC-0B471AB7-PEAK-LOAD-TEASTING.html, last visited on February 7, 2021

11. Spike Test

Definition: a spike test is designed to issue a sudden increase or decrease in load to the SUT [Guru99, 2014].

Objectives: determine the behavior of a software application when it receives extreme variations in traffic, both when traffic increases and when traffic decreases. "The purpose of a spike testing is to determine the recovery time after a spike of user load. It is performed to estimate the weakness of an application" [Guru99, 2014];

Expected Input: as per the peak-load test;

Expected SUT Maturity: as per the peak-load test;

Expected Preliminary Performance Tests: as per the peak-load test;

Execution Environment Requirements: as per the peak-load test;

Execution Process: no specific ones;

When Starting the Test: as per the peak-load test;

During the Test: ensure the SUT is experiencing a spike by monitoring the system's resources utilization and system's performance metrics;

Suspension Criteria: as per the peak-load test;

After the Test: as per the peak-load test.

Test Deliverables: no specific ones;

Metrics: as per the peak-load test;

Quality Results: the quality results report the behavior of the SUT during and after experiencing a spike;

Reports: as per the peak-load test. It is additionally important to report whether or not the SUT experienced a spike in terms of utilized resources and the system's performance metrics.

12. Throttle Test

Definition: a throttle test simulates load with limited bandwidth and connection stability for all or a subset of simulated users[10].

[10] https://abstracta.us/blog/performance-testing/types-performance-tests/, last visited on February 7, 2021

Objectives: analyze the response time of users connected through lower speed networks;

Expected Input: the expected input has to be modeled to simulate degraded network conditions. Traffic shapers are usually employed to simulate the wanted network conditions.

Expected SUT Maturity: as per the scalability test;

Expected Preliminary Performance Tests: as per the scalability test. It is important to ensure the functionalities work in normal conditions before simulating extreme conditions;

Execution Environment Requirements: require and support for degraded network conditions simulation;

Before the Test: Ensure traffic shapers or settings simulating degraded network conditions are correctly working and are in place.

Execution Process: no specific ones;

Before the Test: configure the expected degraded network conditions;

During the Test: change the degraded network conditions to different scenarios to capture the SUT behavior;

After the Test: restore normal conditions for the execution environment.

Test Deliverables: no specific ones;

Metrics: metrics of interest mainly refer to client-side performance metrics, such as the response time and the throughput. Important metrics to monitor for a throttle test are also related to the NET, and in particular metrics on packet loss, network errors, and network bandwidth;

Quality Results: the quality result indicates the behavior of the SUT in dealing with requests received when degraded network conditions are in place;

Reports: the report has detail on all the metrics on interest, often compared with the same metrics computed in normal network conditions.

13. Soak or Stability Test

Definition: a "soak test is intended to identify problems that may appear only after an extended period of time" [Molyneaux, 2014].

Objectives: ensure the SUT does not experience failures and the SUT's performance does not degrade over time when users keep interacting with the system, issuing expected load [Molyneaux, 2014] or under a higher than expected load [Guru99, 2014].

Expected Input: the expected input represents realistic workloads experienced by the SUT during its operation in production, as well as higher than expected workload.

Expected SUT Maturity: as per the scalability test, but usually executed in the final stage of the development before releasing the software to production;

Expected Preliminary Performance Tests: as per the scalability test;

Execution Environment Requirements: represents as close as possible the production environment and is available for the entire test duration, that might span over multiple days;

Execution Process: no specific ones;

During the Test: no specific ones;

Suspension Criteria: the behavior of the SUT has to be monitored to ensure the sustained load does not impact the performance of the same, especially in terms of resource usage. If unexpected behaviors are detected, one might decide to suspend the test execution to avoid wasting resources and time.

After the Test: as per the load test.

Test Deliverables: no specific ones;

Metrics: the metrics of interest look both at client-side and server-side performance factors that could degrade over an extended period of time. Examples of monitored client-side metrics of interest are response time and throughput. Examples of server-side metrics of interest are related to RAM (e.g., because the system could experience memory leaks) and DISK usage (e.g., because the system could saturate the disk space over an extended period of time);

Quality Results: the quality result indicates whether or not the SUT correctly exercises at its full range of use, without failing or causing failure;

Reports: "Problems of this sort will typically manifest themselves either as a gradual slowdown in response time or as a sudden loss of availability of the application" [Molyneaux, 2014], thus the report details relevant aspects to determine the behavior of the SUT over an extended period of time, and present charts of metrics over the entire test duration to evaluate the trends of measured performance indicators. It is also important to provide a correlation of measures from the client- and the server-side over time, for accurate diagnosis of possible issues [Molyneaux, 2014].

14. Exploratory Test

Definition: an exploratory test is designed to learn about the performance of the target SUT and to discover its behavior in different scenarios [Guru99, 2014].

Objectives: learn about the possible performance risks one has to define tests for, and discover how the SUT behaves in different scenarios to better characterize its performance [Guru99, 2014] and defines SLOs.

Expected Input: the expected input has to be modeled to represent wide-spectrum scenarios exploring the performance of the SUT with different workloads and configurations.

Expected SUT Maturity: the exploratory test has no specific requirements in terms of SUT maturity. The functionalities of the SUT objective of testing have to be sufficiently ready for enabling the test execution;

Execution Environment Requirements: needed resources depend on the definition of the exploration tests. Given the explorations process is by nature made in multiple repetitions, proper sizing of the execution environment is needed to allow for parallelization of multiple exploration tests;

Execution Process: no specific ones;

During the Test: no specific ones;

Suspension Criteria: it is important to closely monitor the behavior of the SUT during different experiments while exploring the performance, to avoid resource and time wasted in exploring performance in unstable areas of the performance space of the SUT;

After the Test: as per the load test.

Test Deliverables: no specific ones;

Metrics: defined metrics depend on the executed exploration and the shape of the issued workload. Usually, many performance metrics, both on the client- and the server-side are defined because the scope is collecting information about the SUT's performance;

Quality Results: the quality results indicate how the SUT performs in the different points of the exploration space;

Reports: the report has details on the performance of the SUT in the different points of the performance space. It is common to report charts representing metrics of interest value in different points of the performance space, to offer an overview of how the system behaves.

15. Configuration Test

Definition: a configuration test "is defined as a software testing type, that checks an application with multiple combinations of software and hardware to find out the optimal configurations that the system can work without any flaws or bugs" [Guru99, 2014].

Objectives: determine the configurations of the SUT and its deployment environment where the system works without experiencing failures and guaranteeing the SLA and optimal performance;

Expected Input: the expected input has to be modeled to account for possible SUT's configurations and deployment scenarios;

Expected SUT Maturity: as per the scalability test;

Expected Preliminary Performance Tests: as per the scalability test;

Execution Environment Requirements: as per the exploration test;

Execution Process: no specific ones;

During the Test: no specific ones;

Suspension Criteria: as per the exploration test;

Test Deliverables: no specific ones;

Metrics: defined metrics depend on the defined SLO. It is important to always monitor server-side metrics characterizing the resource utilization, to better identify proper configurations of the SUT;

Quality Results: as per the exploration test;

Reports: as per the exploration test.

16. Benchmark Performance Test

Definition: a benchmark test is designed to assess the performance of multiple versions of a system or similar systems against a set of metrics designed to assess the quality of a service [Guru99, 2014].

Objectives: "measures a repeatable set of quantifiable results that serves as a point of reference against which products/services can be compared. The purpose of benchmark testing results is to compare the present and future software releases with their respective benchmarks" [Guru99, 2014].

Expected Input: the expected input has to be modeled to account for all the relevant usage scenarios of the SUT;

Expected SUT Maturity: as per the scalability test;

Expected Preliminary Performance Tests: a benchmark can include multiple usage scenarios, thus all the test types representing the included performance test scenarios have to be executed before a benchmark test to ensure the system is ready for being characterized using a benchmark. Additionally, all the tests as per the scalability test have to be executed;

Execution Environment Requirements: as per the exploration test. Repeatability of experiments is particularly relevant for benchmark performance test;

Execution Process: no specific ones;

During the Test: no specific ones;

Suspension Criteria: as per the exploration test;

Test Deliverables: no specific ones;

Metrics: Defined metrics have to characterize the SUT's performance and be stable and repeatable to guarantee the possibility to be used as reference points. The defined metrics depend on the target SUT, and are often custom to the specific SUT type, as it is in the case of WfMSs [Ferme et al., 2019];

Quality Results: the quality results report the quality of a target system when compared with the results of other systems, or other versions of the same system, measured using the same performance benchmark;

Reports: the report describes the quality of the system according to the measured metrics as part of the benchmark, in absolute terms and when compared with measurement collected by executing the same benchmark on other versions of the same system, or similar systems. It is common to report results using a ranked list of systems, or system versions, for the different metrics to facilitate the positioning of the obtained results when compared with other results.

17. Acceptance Test

Definition: an acceptance test is designed to validate the performance of the SUT against defined acceptance criteria[11].

Objectives: assess changes applied to the SUT does not impact the defined acceptance criteria for the SUT to be deployed in production;

Expected Input: it depends on the defined acceptance criteria and the types of workload they require for being verified;

Expected SUT Maturity: as per the scalability test;

Expected Preliminary Performance Tests: as per the scalability test;

Execution Process: as per the regression test;

Test Deliverables: no specific ones;

Metrics: the metrics of interest are the one included in the acceptance criteria;

Quality Results: the quality result indicates whether or not the SUT passes the defined acceptance criteria;

Reports: the report lists all the defined acceptance criteria and whether the SUT passes them or not. It is common to also report additional performance metrics and details, especially related to the failing acceptance criteria.

18. Capacity or Endurance Test

[11] https://www.istqb.org/, last visited on February 7, 2021

Definition: a capacity or endurance test assesses the system can perform as expected over an extended period of time, at its limit of users interacting with the same [Molyneaux, 2014].

Objectives: ensure the SUT performs as expected over an extended period of time when it is tested at its limit. One objective of the capacity test is also to better identify the operating limits. One of the primary focus of endurance testing is to check for memory leaks [Guru99, 2014].

Expected Input: as per the soak test;

Expected SUT Maturity: as per the soak test;

Expected Preliminary Performance Tests: as per the scalability test. Additionally, also the *Soak or Stability Test* has to be executed being similar to a capacity test but focusing also on normal operating conditions for the SUT;

Execution Environment Requirements: as per the soak test;

Execution Process: no specific ones;

During the Test: no specific ones;

Suspension Criteria: as per the soak test;

After the Test: as per the load test.

Test Deliverables: no specific ones;

Metrics: as per the soak test. Additionally, low-level systems' metrics are also monitored to identify possible misbehavior of the system over time. Example of metrics measure the number of open/closed connections on the NET level, and open/closed connections to other systems (e.g., to DBMS);

Quality Results: the quality result indicates whether or not the SUT correctly exercises at its full range of use while operating at its limit;

Reports: as per the soak test.

19. Live-traffic or Canary Performance Test

Definition: a live-traffic or canary test is designed to issue loads to the SUT while it is deployed in the production environment, to assess the SUT is ready for handling actual users' load [Schermann et al., 2018].

Objectives: ensure the SUT is ready to be exposed to end-users;

Expected Input: as per the load test;

Expected SUT Maturity: the SUT must be ready to be deployed in production;

Expected Preliminary Performance Tests: as per the scalability test;

Execution Environment Requirements: enable execution of live testing by isolating multiple versions of the SUT deployed in parallel on the production infrastructure [Schermann et al., 2018; Veeraraghavan et al., 2016].

Execution Process: no specific ones;

Before the Test: ensure one or more versions of the SUT are deployed in production;

During the Test: simulate traffic or incrementally mirror more and more live-traffic to the target version of the SUT;

Suspension Criteria: ensure there are no impacts of the production performance of the SUT for the versions that are directly utilized by the user, otherwise quickly suspend the test;

Before Concluding the Test: validate the new version of the SUT is ready for end-user traffic;

After the Test: if the new version of the SUT is ready for end-user traffic, switch the user-facing version to the new version.

Test Deliverables: no specific ones;

Metrics: the metrics of interest are looking at different aspects of performance both from the client-side and the server-side. They are defined according to the type of tested SUT to better characterize its performance behavior [Veeraraghavan et al., 2016];

Quality Results: the quality result indicates whether or not the new version of the SUT can or has successfully been exposed to the end-users;

Reports: the report has to report details on the actual workload that has been issued to the new version of the SUT, the behavior of other versions of the SUT deployed in production and exposed to end-users and whether or not the new version of the SUT has been exposed to end-users or is ready to be exposed. Live metrics collected from actual production usage

of the new version of the SUT after exposing it to end-users have to be reported as well.

20. Chaos Test

Definition: chaos testing deliberately introduces failures into the SUT or its underlying infrastructure to ensure it can deal with the failures[12].

Objectives: ensure the SUT can correctly operate even in case of controlled failures introduced into the SUT or its underlying infrastructure;

Expected Input: the expected input has to be modeled to introduce failures into the system and test the same with expected workload;

Expected SUT Maturity: as per the live-traffic test;

Expected Preliminary Performance Tests: as per the scalability test;

Execution Environment Requirements: enable the possibility to introduce failures by supporting interaction with the infrastructure;

Execution Process: no specific one;

During the Test: no specific one;

Suspension Criteria: ensure chaos actions are not degrading the system's performance to a point it becomes unstable and further actions are not possible;

Test Deliverables: no specific ones;

Metrics: as per the exploration test. Chaos testing is performed similarly to exploration testing, but different actions are executed;

Quality Results: the quality results indicate the effect of each action executed during the test on the SUT's performance;

Reports: the report details how the different monitored metrics are impacted by the different actions executed during the chaos test. It is also important to provide a correlation of measures from the client- and the server-side before, during, and after executing actions introducing failures to better diagnose possible problems.

[12] https://netflixtechblog.com/the-netflix-simian-army-16e57fbab116, last visited on February 7, 2021

21. Breakpoints Performance Test

Definition: a breakpoints performance test "helps to find out the breaking point of the application or server from performance testing perspective"[13].

Objectives: identify the SUT breakpoints by purposefully loading the system over its limit and discover as many breakpoints as possible;

Expected Input: as per the stress test, but specifically targeting the identification of breakpoints;

Expected SUT Maturity: as per the load test;

Expected Preliminary Performance Tests: as per the peak load test;

Execution Environment Requirements: as per the peak load test;

Execution Process: no specific ones;

When Starting the Test: as per the scalability test;

During the Test: attempt to maximize breakpoints discovery by pushing the SUT to the limit;

Suspension Criteria: as per the stress test;

After the Test: as per the load test.

Test Deliverables: no specific ones;

Metrics: as per the stress test;

Quality Results: the quality results report the behavior of the SUT at its breakpoints;

Reports: as per the peak load test.

22. Failover or Recovery Test

Definition: a failover or recovery test "validates a system's ability to be able to allocate extra resource"[14].

[13] https://www.perfmatrix.com/application-break-point-test/, last visited on February 7, 2021

[14] https://qa-platforms.com/what-is-failover-testing/, last visited on February 7, 2021

Objectives: overload the SUT over its limit and verify the system can allocate additional resources to guarantee it continues to operate without too many user-facing issues;

Expected Input: as per the stress test;

Expected SUT Maturity: as per the load test;

Expected Preliminary Performance Tests: as per the elasticity test. If elasticity testing can not be performed on the SUT, then as per the scalability test;

Execution Environment Requirements: as per the scalability or elasticity test;

Execution Process: no specific ones;

When Starting the Test: as per the scalability test;

During the Test: no specific ones;

Suspension Criteria: as per the scalability test with particular focus on the ability of the SUT to allocate extra resources and avoid failures;

After the Test: as per the load test.

Test Deliverables: no specific ones;

Metrics: the metrics of interest are mainly related to the resource utilization of the SUT and metrics characterizing incurring failures as the one identified for the stress test;

Quality Results: the quality result indicates the behavior of the SUT in adjusting its resource utilization to avoid propagating incurring failures under high load;

Reports: as per the stress test.

23. Resiliency or Reliability Test

Definition: a resiliency or reliability test validates the "ability of a solution to absorb the impact of a problem in one or more parts of a system while continuing to provide an acceptable service level to the business"[15].

[15] https://www.ibm.com/developerworks/websphere/techjournal/1407_col_nasser/1407_col_nasser.html, last visited on February 7, 2021

Objectives: induce failures in the SUT and verify the system can guarantee it continues to operate without too many user-facing issues;

Expected Input: as per the chaos test;

Expected SUT Maturity: as per the load test;

Expected Preliminary Performance Tests: as per the elasticity test. If elasticity testing can not be performed on the SUT, then as per the scalability test;

Execution Environment Requirements: as per the chaos test;

Execution Process: no specific one;

During the Test: no specific one;

Suspension Criteria: ensure introduced failures are not degrading the system's performance to a point it becomes unstable and further actions are not possible;

Test Deliverables: no specific ones;

Metrics: the metrics of interest mostly refer to client-side metrics, because the objective of the test is to analyze the impact of failures on users. It is also important to provide server-side metrics, especially to assess the behavior in case the SUT fails to adjust to introduced failures;

Quality Results: the quality results indicate the effect of introduced failures on the SUT's performance;

Reports: the report details how the different monitored metrics changes when failures are introduced to the system. It is important to report whether or not the system smoothly reacts to introduced failures, without experiencing issues. As per the chaos test, it is important to provide correlation charts between client- and server-side metrics when failures are introduced.

24. Snapshot-load Test

Definition: a snapshot-load test evaluates how the GUI of the SUT behaves under load. "A typical snapshot test case renders a GUI component, takes a snapshot, then compares it to a reference snapshot file stored alongside the test"[16] and fails if they are different.

[16] https://jestjs.io/docs/en/snapshot-testing, last visited on February 7, 2021

Objectives: ensure the GUI of the SUT correctly renders under load;

Expected Input: as per the load test. Additionally, reference snapshots have to be provided;

Expected SUT Maturity: as per the load test. It is important the SUT is ready to experience the expected production load;

Expected Preliminary Performance Tests: as per the load test. Additionally, the *Load Test* has to be executed as well to ensure the SUT can manage the load;

Execution Environment Requirements: as per the load test. Additionally, it is important to collect snapshots on all the possible execution environments in which the GUI is expected to be accessed from. For doing so it is needed that the execution environment enables the collection of snapshots. In case of deployment on a mobile client it is often needed to enable debugging functionalities to guarantee snapshots can be collected;

Execution Process: no specific ones;

Before the Test: ensure snapshots can be collected;

When Starting the Test: as per the load test;

During the Test: no specific ones;

Suspension Criteria: as per the load test;

After the Test: as per the load test.

Test Deliverables: no specific ones;

Metrics: metrics of interest provides client- and server-side details on the issued load to characterize the same;

Quality Results: the quality results indicate whether or not the GUI renders correctly under different load and deployment contexts;

Reports: the report details the different monitored metrics and additionally report comparisons on snapshots taken during the test with the reference snapshots and highlight differences across all the deployment contexts in which the test is executed.

25. Volume or Flood Test

Definition: a volume or flood test issues a huge volume of data to the SUT or to the DBMS the SUT relies on [Guru99, 2014].

Objectives: ensure the SUT's performance is not impacted by a huge amount of data issued to the same or stored in DBMSs the system relies on;

Expected Input: the workload is expected to issue a huge volume of data, thus complex and big payloads have to be defined for the interaction with the SUT;

Expected SUT Maturity: as per the scalability test;

Expected Preliminary Performance Tests: as per the snapshot-load test;

Execution Environment Requirements: the execution environment has to be able to manage the huge volume issued to the SUT and potentially stored by the SUT on the hosting infrastructure;

Before the Test: ensure expected data is correctly loaded into the system.

Execution Process: no specific ones;

Before the Test: load the data the test is expected to find in the target system as a starting point for the test;

During the Test: no specific ones;

Suspension Criteria: it is important to monitor the behavior of the SUT when a huge amount of data is issued. If the system starts to experience unexpected behavior, one might decide to suspend the test.

After the Test: as per the load test.

Test Deliverables: no specific ones;

Metrics: the metrics of interest look at both client- and server-side factors potentially influenced by a huge amount of data handled by the SUT. Relevant metrics are for example the response time, that could be impacted by the increasing amount of data, and DISK related metrics to ensure the disk is efficiently used by the SUT ad not overloaded;

Quality Results: the quality result indicates whether or not the SUT properly handles a huge volume of data;

Reports: the report has to indicate how the system handles the huge amount of data issued during the test. It is important to report the metrics of interest along with statistics on the size of the issued data or data stored in DBMSs the SUT relies on. This helps to identify limits in the system performance in handling the data.

4.4 Performance Tests Goals and DevOps

Usually, in CSDL, users define performance tests they might want to continuously see working, and that they automate in terms of automating the analysis and the process to collect the data for that analysis. Then they define another set of performance tests that is not always continuously executed, based on a model they have (i.e., requirements, design, implementation diagrams, etc.) to explore the system, learn from it and maybe decide to automate other analyses to be able to communicate derived information in a better way.

4.4.1 Concepts

In CSDL, software gets developed and released in continuous iterations. In the early stage of software development, the iterations are usually slow in building up until a first minimum viable release is reached. From this moment on, multiple iterations are executed according to end-users' feedback and business needs. The more mature the software is, the fewer iterations are expected, and usually, the iterations velocity slows down as well.
In this context, the software is built by pushing the code to a Version Control System (VCS), and many developers work on the software at the same time. To enable this kind of development, different processes exist, one of them being the GitFlow [Driessen, 2010] one. GitFlow, as well as other similar development processes, usually develops software in different feature branches, usually named "feat-<FEATURE_NAME_ID>". Feature branches get merged to a "development" branch through a pull request when they are considered completed and successfully tested. The "development" branch represents the code under development and the code that is expected to be moved in production next. Pull requests are represented as development branches as well, usually named "pr-<FEATURE_NAME_ID>", and always represent two versions of the software under development: the "HEAD" version and the "MERGE" version. The "HEAD" represents exactly the version of the code present in the branch from which the pull request has been opened. The "MERGE" version

represents the code the developer obtains after merging the "HEAD" code with the destination branch of the pull request, in our example the "development" branch. When the new version of the code is considered ready for production, usually branches named "release-<RELEASE_NUMBER>" are created and code is merged to the "master" branch, representing the production code, through a pull request. Pull requests are also used to merge code fixing production bugs, usually pushed in branches named "hotfix-<HOTFIX_NAME>", to the production branch.
Every action described above, generate some events coming from different entities, and in CSDL, it is important to be able to control which test, or set of tests, to execute according to the type and the source of the event. This is important because usually different test types, or different configurations for similar test types, are executed according to the event. It is important also because, as presented, according to the branch where the code resides, the target deployment environment is different. In CSDL, usually code is continuously built by Continuous Integration System (CIS), such as Jenkins[17] . CIS are linked to the VCS systems in such a way they are collecting all the mentioned events, and based on those events the defined workflow processes building the software execute specific steps. Examples of steps that are often executed are a step to build the software, a step to test it and a step to build a deployable artifact out of the code. In some contexts, as part of CSDL, are defined also workflows and processes to continuously deliver the software to the different deployment environments. These are called Continuous Delivery System (CDS), and one example is CloudBees CD[18] . Jenkins is also sometimes used as CDS system, although it is not built specifically for this use case.

4.4.2 CSDL and Performance Tests Execution

Performance tests are part of the CSDL and are integrated with the same according to different criteria. In Fig. 4.1 we report an example of integration for some of the performance tests part of the catalog presented in Sect. 4.3. Figure 4.1 represents a CICD pipeline, integrating Smoke tests, Load tests, Acceptance tests, Regression tests, Canary tests, and Live tests. The tests are executed in different steps of the pipeline, and some of them are executed in parallel. Based on the continuous execution of workflows similar to the one in the example, a user can decide to define quality gates and disable some steps

[17] https://www.jenkins.io/, last visited on February 7, 2021

[18] https://www.cloudbees.com/products/continuous-delivery, last visited on February 7, 2021

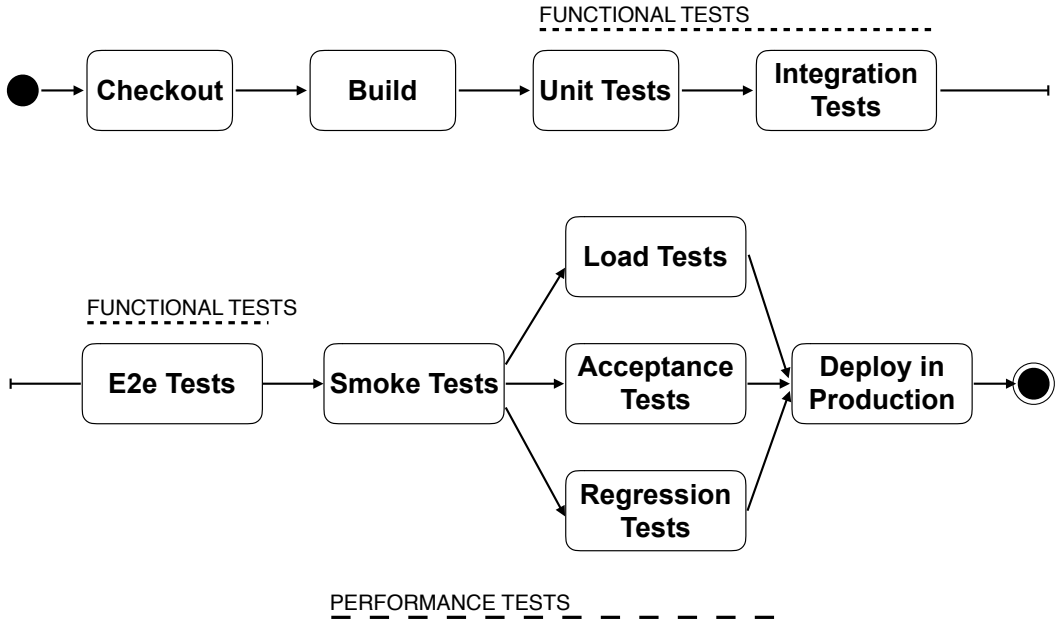

Figure 4.1. CSDL: Example of Performance Test Integration

if they are not leading to effective results (e.g., performance tests that always experience a stable performance, or that are not relevant for an early catch of performance problems). In the following sections, we present different selection criteria for deciding how and when to integrate performance tests execution as part of the CSDL. Tests suggested to be executed according to different criteria might overlap.

4.4.3 CSDL and Performance Tests Selection

We define different criteria for deciding how to integrate performance test execution as part of the CSDL. They are related to the factors we identified in Sect. 4.2.1 for classifying the different performance test types in the catalog. The criteria are:

1) base on manual scheduling. Some test types, for example, the exploratory and the configuration test, are usually scheduled for execution manually. The reason is that those are tests the user defines for exploring the performance of the SUT or assess the right configuration to set to ensure required performance behavior respectively. These types of tests are important because they allow the user to set SLO, based on SLAs;

2) based on the development flow and developed entities. Tests selected according to this criteria are usually integrated with CICD and scheduled according to events generated during the development flow and the software entities being developed;

3) based on the SUT maturity level. Some tests can be executed only when the SUT reaches certain maturity levels in terms of its development;

4) based on the expected behavior of the clients interacting with the SUT, as well as where the SUT is deployed and its runtime context. This criterion impacts the types of tests to be selected for the integration with the CSDL because different contexts require different test types.

When integrating performance tests as part of the CSDL, it is fundamental to continuously improve the SUT based on the results obtained from test execution. This has to happen during development iterations, to ensure the effort, time, and resources spent in executing performance test has the right impact in improving the SUT.
Deciding which actual tests or test suites to execute is out of the scope of this work. It is usually not feasible to always execute all the tests defined during the CSDL, but a subset of the same shall be selected to cover the most important performance aspects of the SUT. There are different approaches to select tests to be executed, as presented in Sect. 3.2.3.

Manually Scheduled Performance Tests

The test we identify as being usually scheduled manually are:

1) the exploration test;

2) the configuration test;

3) the soak (or stability) test;

4) the capacity (or endurance) test.

Exploratory tests are defined and executed by users exploring the performance of the SUT, to learn about its behavior. Exploratory testing is very important in CSDL because provides the users with data about the performance of the developed system while the development is ongoing. Configuration tests are defined when the SUT implements a viable set of features to be considered ready for the next phases bringing it to production. The configuration tests

are defined to identify the best configuration for the SUT according to SLO and the defined workload. They are usually scheduled manually because they are complex and time-consuming, thus are only executed when needed. Soak or stability tests, as well as capacity or endurance tests, similar to the configuration tests, are complex and time-consuming, thus are usually scheduled for execution manually.
The same tests can also be scheduled as part of CICD pipelines, although the exploration test is rarely integrated because it is meant to be executed from a user testing her performance assumptions towards the SUT.

Selection Based on the Development Flow

During the development flow, many events are generated, as presented in Sect. 4.4.1, according to the followed development process. In the following we propose how different events, happening in different moments of the development flow, shall be mapped to different performance test types, that are usually scheduled as part of the CICD pipelines.
Considering the following development branches a software following the Git-Flow might define (* represents a generic string): 1) master (production); 2) release-*; 3) development; 4) feature-*; 5) hotfix-*. , and the following events generated as part of the CSDL: 1) push of a set of commits; 2) opening of a pull request; 3) merging of a pull request; 4) opening a branch; 5) SUT deployment.
In the following, we discuss how different types of performance tests shall be scheduled, and in Tab. 4.1 we summarize our proposal.
When developing a *feature-**, exploratory performance tests are executed to build performance knowledge about the SUT. Every time a new *push of a set of commits* happens on *feature-** and *hotfix-** branches, we advise scheduling unit performance tests and smoke tests on the committed code. Unit performance tests and smoke tests can be executed in a fairly limited amount of time, and provide initial insight on the performance of the SUT.
When *opening a pull request* from a *feature-** to the *development* branch, we suggest to execute: 1) smoke tests; 2) performance regression tests, to ensure newly developed features comply with regression criteria; 3) sanity tests; 4) load tests. Optionally, according to the actual objectives of the changes committed as part of the feature, scalability tests shall be also scheduled for execution. All of the above-mentioned tests shall be executed on the code base representing the new code to be merged part of the *feature-** branch (i.e., on the "HEAD" of the pull request).

Branches	Events	Scope	Scheduled Performance Tests
feature-* feature-* hotfix-*	- push of a set of commits	System Committed Code	Exploratory Unit Smoke
FROM feature-* TO development	opening of a pull request	Committed Code	Smoke Regression Sanity Load Scalability (Optional)
FROM development/hotfix-* TO master (production)	merging of a pull request	System	Stress Spike Acceptance Soak Elasticity (Optional)
development	push of a set of commits AND opening a release-* branch	System	Configuration Peak-load
master (production)	push of a set of commits	System	Capacity Volume Breakpoint (Optional) Failover (Optional) Resiliency (Optional)
development	SUT deployment IN staging	System	Chaos
master (production)	SUT deployment IN pre-production and production	System	Live-traffic Chaos

Table 4.1. Performance Tests Selection Based on the Development Flow

When a pull request is ready to be merged, just before *merging of a pull request* from the *development* branch to the *master (production)* branch, or from a *hotfix-** branch to the *master (production)* branch, we advise to execute: 1) stress tests; 2) spike tests; 3) acceptance tests; 4) soak tests. Optionally, according to the SUT, elasticity tests shall be also scheduled for execution. The above-mentioned tests have to be executed on the "MERGE" of the pull request. This way, we ensure the system resulting from merging the newly developed code, with the code currently running in production performs as expected before actually updating the codebase.
When *opening a branch* named *release-** and when a *push of a set of commits* happens on the *development* branch (e.g., after completing a *merge of a pull request*), we propose to execute the following tests: 1) configuration tests; 2) peak-load tests. Additionally, other tests can be scheduled, to deeper assess the performance of the system, for example during the night (nightly executions). When the code is merged on the *development* branch, usually the resulting version of the SUT is deployed in a staging environment, chaos testing can be executed.
When the *push of a set of commits* happens on the *master (production)* branch, we propose to execute the following tests: 1) capacity tests; 2) volume tests. Optionally, according to the SUT, breakpoint, failover, and resiliency tests shall be also scheduled for execution. The master (production) branch is expected to be deployed in pre-production or/and production. When that happens, live-traffic tests can be performed, as well as chaos testing if it is safe to be executed.
The proposed tests to be executed must be considered a proposal. Not all the tests shall be scheduled for all kind of SUTs. In proposing the different tests to be executed we mainly refer to the target SUT of our work, namely RESTful Web services and BPMN 2.0 WfMSs.

Selection Based on the SUT Maturity Level

The software development process is comprised of many different phases. The different phases move the software from low to high level of maturity in terms of its functional and non-functional completeness according to the requirements. The maturity of the SUT has an impact on the test that shall be executed.
The levels of SUT maturity we identify are:

1) prototyping, the first stage when the initial code of the SUT is added to the code base;

2) features, the SUT integrates an initial set of under-development features;

3) development, the first development version of the SUT is ready, integrating different features according to the requirements;

4) alpha, an initial potential release version that is made available for testing, typically by employees of the company that is developing it;

5) beta, an initial potential release version that is made available for testing, typically by a limited number of users outside the company that is developing it;

6) release candidate, a candidate release version that is made available for testing to a large audience of users outside of the company that is developing it;

7) release, a release version for the SUT ready to be deployed in production.

During prototyping, usually, no performance tests are executed. While developing features, unit performance tests can be executed on stable functionalities. Unit performance tests at this stage are usually defined on methods and classes for object-oriented systems. It is common also to execute smoke tests at this level of maturity. When the first development version is ready, load, and scalability tests shall be executed against the SUT. When users start to test the SUT, starting from the alpha stage, it is important to define and execute also: recovery, breakpoints, resiliency tests. These are important because ensure the collected feedback from the users is reliable and effective. It is, e.g., not advisable to release the software to external users (beta stage) without measuring software resiliency and recovery in case of failure, because of potentially missing control of SUT behavior during the tests. While moving the SUT towards a release, it is important also to execute a capacity test targeting the entire system. This is usually done at the beta or release candidate stage. The release candidate version is also the target of configuration tests, to ensure the definition of the right configuration before moving the software to production. The same version, as well as the production version at the release stage, undergoes volume testing.
The previous maturity levels apply to the entire SUT, but also to newly added features. In CSDL the SUT usually reaches the release stage, while new versions of the same SUT are already under development and reached one of the other maturity levels in a continuous cycle.

Selection Based on the SUT Client, Deployment and Runtime Contexts

Different software is deployed in different contexts. The contexts set requirement in terms of the expected behavior of the clients interacting with the SUT, as well as the SUT deployment and runtime contexts set requirements towards the test types that shall be executed.
Two tests highly characterized by the client context are the throttle and the snapshot-load tests. These two types of test are mainly scheduled for mobile applications or embedded software. The throttle test ensures the SUT is behaving correctly when network bandwidth is unstable, while the snapshot-load test is executed to ensure the user interface is shown consistently to the user under different load profiles. The latter is also executed when the SUT is expected to be accessed by many different clients having different characteristics in how they handle the user interface.
For what concern the SUT deployment environments, the Cloud computing context has quite an impact on the test types to be scheduled for execution. In Cloud computing environments, it is very important to execute elasticity, scalability, stress, and spike tests. The same tests are mostly executed also in other environments, but for the Cloud computing environment, they are required to successfully empower the features of the platform on which the SUT is deployed to.
The SUT runtime contexts have an impact on the scheduled test as well. When the SUT is deployed on a virtual machine, for example, the JVM, it is also very important to execute benchmarks on the JVM itself to be sure is correctly configured to operate the systems deployed on top of it.

4.5 Concluding Remarks

In this chapter, we present the automation-oriented performance test catalog, realized by looking at contexts and factors defining performance test types. The presented catalog answers R.G. 1. We first illustrate the different contexts, i.e. the SUT and the client context, and discuss the factors impacting the performance tests automation process. We then present a template for the proposed automation-oriented catalog, and we present all the identified performance tests using the proposed template. After presenting the catalog, we introduce how DevOps, and development and software release processes applied in DevOps contexts are mapped to performance test selection in CSDL. This contributes towards R.G. 4, and provides an analysis of a proposed mapping of CSDL events to selected performance test suites supported by the contributed

DSL, setting the ground for the related DSL discussed in Sect. 5.5.

Chapter 5

BenchFlow DSL: Declarative Specification of Performance Test Automation Processes

5.1 Introduction and Requirements

In this chapter we tackle the R.G. 2, presenting a DSL enabling declarative specification [Lämmel, 2018] of performance test automation processes. The DSL has been named BenchFlow DSL after the BenchFlow project presented in Sect. 1.5. In our work, we look at automating the execution of performance tests issued against systems' APIs, particularly REST APIs, exposed to the users. In particular for the case of RESTful Web services and BPMN 2.0 WfMSs.

We target tangible software systems and thus focus on activities and roles where actual code is written, tested, and operated according to the defined requirements, design, and architecture. For this reason, our main targeted users are developers, software testers, quality assurance engineers, DevOps engineer, and operations engineers. Developers, including software testers, develop and test the part of the software they are responsible for. Quality assurance engineers usually supervise the quality of the produced artifacts and thus are involved in all the activities related to testing, important for guarantying the delivery of high-quality new releases. DevOps and operations engineers are responsible for the deployment, use, and run-time monitoring of the system, thus contribute to executing tests and improving them with real-world knowledge about software behavior. All the three mentioned user profiles, when working in a context where the DevOps approach is applied, are expected to be aware

of the deployment environment [Zhu et al., 2016; Spinellis, 2016]. Additionally, to enable the DevOps practices they are also expected to know about packaging and deployment technologies, e.g., virtual machine images, and/or Linux containers (e.g., Docker [Docker Inc., 2013]).
According to the literature, a DSL for providing the specification of goal-driven performance tests should provide at least the specification of load functions, workloads, simulated users, test data, testbed management, and analysis of performance data [Bernardino et al., 2014; Westermann, 2014]. The proposed DSL provides the mentioned features and adds a goal-oriented, and declarative specification of performance intents and performance test automation process as well.
Given the described context, we identified the following requirements the proposed DSL embraces:

1) allows a goal-oriented specification of performance intents;

2) enables the definition of load functions, workloads, simulated users, test data, testbed management, performance data collection, and analysis;

3) supports the configuration of the entire performance test automation processes, and the evaluations to be made during such process;

4) enables human- and machine- readable declarative implementation of the specification model that can be serialized and stored in a VCS;

5) provides syntactic and semantics validation of the specification.

Users of the DSL can specify their performance intents by relying on an expressive goal-oriented language, where standard (e.g., load tests) and more advanced (e.g., regression tests, and configuration tests) performance tests can be specified starting from templates or scratch. Users can then update the tests according to changing requirements, or dispose of them in favor of new specifications, by manually or programmatically updating the test specification. When those tests are committed alongside the code in a VCS, if a developer adds or changes an endpoint then they can also update the performance tests in the same pull request.
We opt for a declarative DSL, so users can rely on a domain model closer to the performance testing terminology. According to the literature, a DSL in the context of DPE should "*enable the formulation and answer of performance-relevant questions and goals for a software system in a human-understandable manner*" [Walter, 2018]. In the proposed approach, the code defining how

to execute the test is generated from the declarative specification (i.e., the DSL) and it is built in the system interpreting the specification. The users do not necessarily need to know how the actual performance test execution is implemented. They only need to be able to define performance activities and control the configuration of the execution of the performance tests. By reducing the needs for the users to write code, the responsibility of translating the business domain into a program shifts from the programmer to the interpreter of the DSL. This has the benefit that the translation is consolidated in one single point (the interpreter of the DSL) and can be verified or even proven to be correct.

5.2 Concepts

To better follow the DSL meta-model and mode entities and structure, we define some useful concepts. The proposed DSL is part of our contribution towards a DPE approach for performance testing, thus some important concepts to be defined belong to DPE.
In DPE, the user is expected to define the intent of the performance testing activity in a declarative manner, indicating the specification of the expected test execution. This specification is then interpreted by a system, able to configure itself and define a plan to reach the status defined in the declarative specification. In DPE, the intent is the rationale for how a user wants to run their performance test. Moving from concrete performance test execution to motivating reasons to arrive at the performance test intent often requires several layers of abstraction [Beyer et al., 2016, Chapter 18]. Consider the following series of generalizations as an example of the abstraction process needed to define a test in a declarative manner:

1) "I want to issue the specified load to a service A configured to use N CPU cores, M GBs of RAM and deployed in two replicas on a cluster X". This is an explicit performance test request, where we detail the configuration of the SUT and where the SUT is expected to be deployed. The question here is, why do we need the specified amount of CPU and RAM, and why the system needs to be deployed exactly in two replicas on cluster X?;

2) "I want to issue the specified load to a service A configured to use N CPU cores, M GBs of RAM and deployed in two replicas on any of the available clusters". This request enables more degrees of freedom and is

potentially easier to fulfill providing more value to the user executing it, although it does not explain the reasoning behind its requirements. Part of the previous question remains, why do we need the specified amount of CPU and RAM and exactly two replicas?;

3) "I want to issue the specified load to a service A configured such that it can handle the specified load, and I want the service to be deployed in two replicas on any of the available clusters". Much more flexibility is provided with the previous question and it is possible to better understand why the user is requesting that a specific amount of resources in the first example. She has to guarantee to be able to sustain the issued load. Part of the previous question remains, why do we need exactly two replicas?;

4) "I want to issue the specified load to a service A configured such that it can handle the specified load guaranteeing the $95th$ percentiles of the response time to be less than $300ms$ at any given point in time". This is a more abstract requirement specification and becomes clear the user is asking to test the service such that she can guarantee a specific requirement. Additionally, the request to have two replicas is also referring to the need that the requirement has to be guaranteed at any given point in time, e.g., also during version updating, thus the requirement of having the service replicated.

The actual CPU and RAM amount requested, the selection of the cluster, and the requested number of replicas specified in previous questions are a guess on what the user thinks might satisfy her requirement, but it is not the intent of the test she wants to execute. The fourth question is the actual one specifying the intent. At this level of abstraction, a DPE approach can enable the definition of the requirement, and then it can take care of defining the actual test execution plan needed to reach such a requirement. From the different levels of generalization, it is clear that to actually being able to define a plan to reach the specified requirement, the approach has also to take into consideration and embrace the possibility to define and control all the details expressed in questions 1-3. The proposed DSL embraces the expressed needs and propose a meta-model accommodating the needs of many of the performance tests presented in Chap. 4.
Other relevant concepts are:

a) *load driver*: the software component built to automatically issue the load specified as part of the test against the SUT;

b) *SUT deployment descriptor*: the deployment specification and its configuration for the SUT, implemented relying on a deployment descriptor language. One example of deployment descriptor language we use as part of this work is Docker Compose [Docker Inc., 2015].

c) *test bundle*: an artifact collecting all the relevant data to execute the declarative performance test submitted by the user, such as test specification, test data, and the SUT deployment descriptor;

d) *test suite bundle*: an artifact collecting all the relevant data to execute the declarative performance test suite submitted by the user, such as the test suite specification, test specification for tests included in the test suite, test data, and the SUT deployment descriptor for the different tests;

e) *experiment*: a well-defined instance of a performance test, executed during the processes of executing the declarative performance test according to the goal specified by the user;

f) *experiment bundle*: an artifact collecting all the relevant data to execute the declarative performance experiment, such as the load driver and its configuration, test data, and the SUT deployment descriptor for all the *trials* of the experiment;

g) *trials*: repeated experiment executions used to collect more precise measurements and to cope with the intrinsic variability of performance;

h) *trial execution framework*: the software solution automating the execution of the trials, and providing load driver deployment and distribution capabilities. Two examples used as part of this work are *Faban* [Faban, 2000], and *JMeter* [JMeter, 1998].

i) *container*: "a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another" [Docker Inc., 2013]. One container technology we rely on as part of this work is Docker.

5.3 BenchFlow DSL Meta-model

The DSL is realized by two main meta-models, the test, and the experiment meta-model. The test meta-model is directly mapped to representing a performance test and expressing the test intent or goal. The experiment meta-model

is mapped to the actual experiments getting executed to reach the goal defined in the test model. The meta-models represent an abstraction on the actual model implementation, omitting details about entities' properties.

5.3.1 Test Meta-model

The test meta-model is realized by different entities, allowing users to specify the test goals and all the configurations needed to control the process of reaching such goal, as well as to control how to deploy the SUT, collect performance data, and compute performance metrics of interest.
The main entities of the meta-model are the *Test*, the *Workload*, the *SUT*, the *Data Collection*, the *Test Configuration*, and as part of the *Test Configuration*, we have the *Goal*, the *Load Function*, the *Termination Criterion*, and the *Quality Gates*, as shown in Fig. 5.1. In Fig. 5.1 we report in blue the central entity of the diagram, as we do for all the following figures in this chapter. Each entity is then connected to other ones, to complete the entire declarative specification of performance test automation processes. We report at meta-model level all the entities of the DSL, as well as their main attributes and relations among the entities, with their cardinalities.
Test - The *Test* is related to many other entities. The most important is the *Goal* entity, where the user can specify her performance intent, by relying on different abstractions made available by the language, or by extending the language itself with custom abstractions. The *Goal* is part of the *Test Configuration* entity, where the user can indicate the *Load Function* (if not part of the *Goal* entity) and its properties (i.e., users, ramp-up, steady-state, and ramp-down), the *Termination Criteria* to control conditions to declare the test

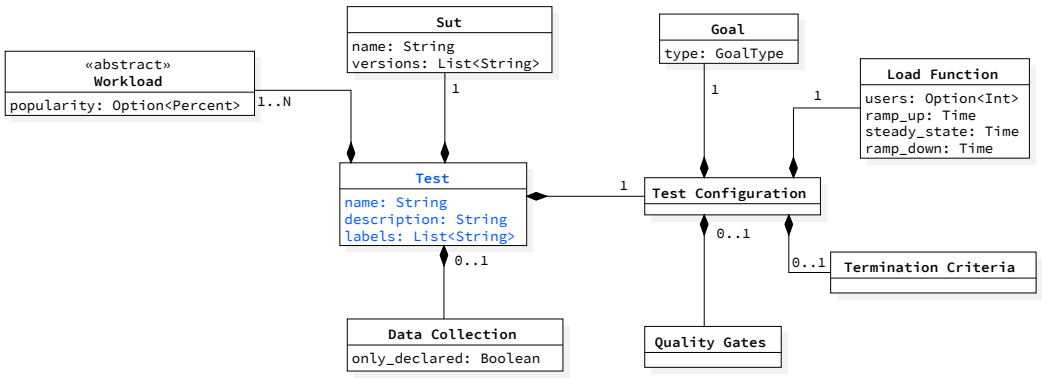

Figure 5.1. DSL: The Test Meta-Model

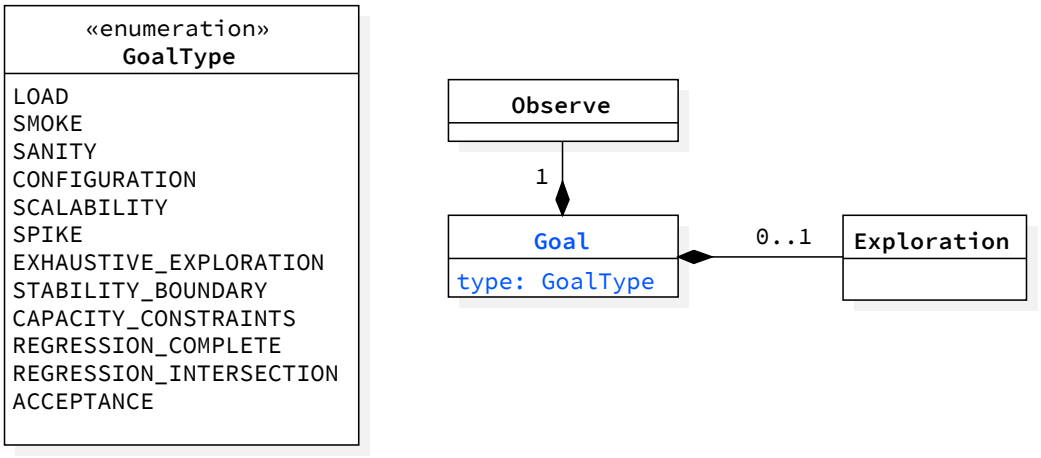

Figure 5.2. DSL: The Goal Meta-Model

completed, and *Quality Gates* to determine if required quality criteria have been achieved or not by the SUT.
Additionally, the user can specify other core performance concepts, such as:

a) the *Workload*, in terms of named sets of operations as well as parameters about the way to mix those sets of operations;

b) details on the *SUT* such as the name and the versions;

c) *data collection services* to enable so that client-side and server-side performance data can be collected.

In the following, we present additional details on the main entities of the DSL, and how the users can specify the different properties of the test definition.
Goal - The *Goal* is part of the *Test Configuration* and is dedicated to declaratively specify the users' performance intent by relying on given performance goals (Fig. 5.2), such as executing a load test or exploring the performance or the stability of the application in given configuration space.
Fig. 5.2 represents the meta-model the user can rely on to specify the goal of the test. The *Goal* is related to the *Observe* and the *Exploration* entities. The supported *GoalTypes* are a subset of the performance test identified and presented in Chap. 4. For some of the performance tests reported in the referenced catalog, for example, the regression test, we have multiple specifications, thus we repeat the performance regression test multiple times as part of the

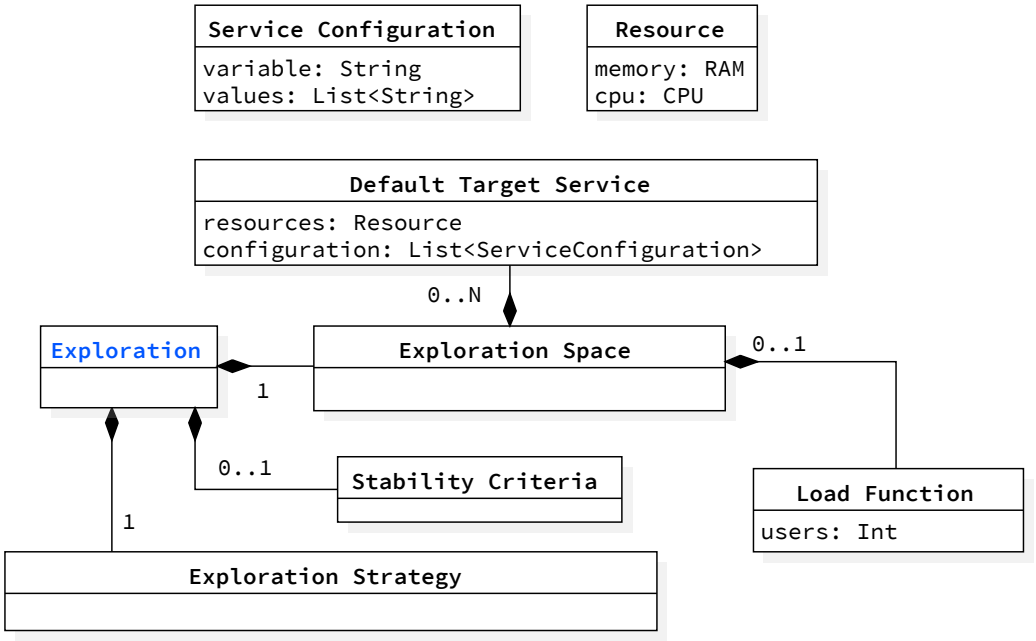

Figure 5.3. DSL: The Exploration Meta-Model

GoalType, in its different facets. This is important to allow the users to clearly state their goals.

Depending on the selected goal it might be necessary to specify other data as well to automate the test execution, e.g., adding an exploration strategy. For example, the *LOAD* goal is a standard load test and does not require additional configuration. The *EXHAUSTIVE_EXPLORATION* on the other end executes multiple experiments by exploring a performance space, thus requires additional specifications under the *Exploration* entity.

Exploration Space - The *Exploration Space* defines the variables that can be changed between experiments and their possible values (Fig. 5.3). The currently available variables can be used for varying the load function, and the configuration of one or more services realizing the SUT or the resources allocated to them. In the current version of the DSL we support CPU and RAM related metrics. For some goals, goal-specific strategies are required, as it is in the case of the *STABILITY_BOUNDARY* test for which we provide *Stability Criteria*

Exploration Strategy - The user decides how to traverse the *Exploration Space*, by selecting an *Exploration Strategy*, as reported in Fig. 5.4. Each strategy determines the order in which different experiments are executed to

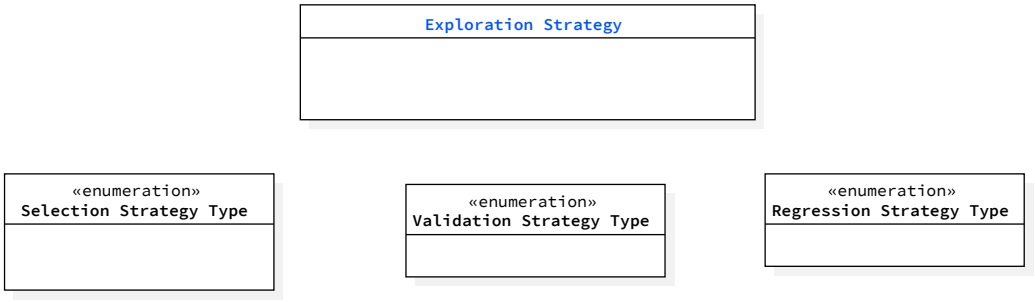

Figure 5.4. DSL: The Exploration Strategy Meta-Model

achieve the goal of the test. Some of the strategies can rely on a statistical sampling approach to reduce the number of experiments required to observe the performance over a representative subset of the exploration space [Westermann, 2014].

Observe - The performance metrics of interest for the test are enumerated within the *Observe* entity (Fig. 5.5). Observed metrics are always optional (as indicated by the cardinality of the relations and the *Option* keyword in the diagram), but at least some are expected to be defined for the performance test to be considered meaningful. Metrics can be observed on the client-side and the server-side, by relying on the collected performance data.

Client-side metrics can be observed on the entire *Workload* and its *Operations*. Server-side metrics can be observed on specific services realizing the SUT. SUT specific metrics can be defined as well.

Termination Criteria - The exploration, and in general, a performance test execution might incur failures and might last for a long amount of time. Thus one or more *Termination Criterion* is used to determine when the test execution process must be terminated (Fig. 5.6).

A termination criterion applies to different entities, namely: the entire test and the navigation of the exploration space, and the different trials of the experiments. At the experiment level, termination criteria can be applied to the *Workload* and on specific services realizing the SUT.

Quality Gates - Quality gates help with integrating the tests in CSDL, by enabling the possibility to express performance requirements for the SUT, for the currently defined test. They declare which are the successful and failure conditions of a test so that these can be checked automatically (Fig. 5.7).

Currently, all the quality gates are verified after the execution of an experiment, thus are applied to each experiment, and consequently to the test if the gates are defined on test configurations or metrics. For some goals, dedicated quality

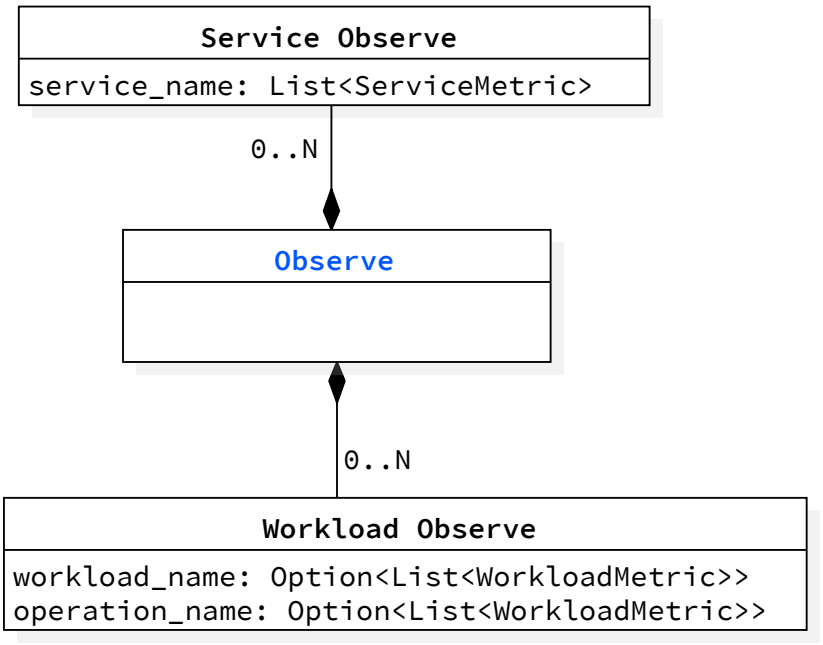

Figure 5.5. DSL: The Observe Meta-Model

gates are required, such as in the case of a regression test, for which we support the specification of a regression condition.
Quality gates complement termination criteria. Quality gates are evaluated after the executions of experiments to determine whether the test succeeded. Termination criteria, on the other end, control and act on the test and experiment execution processes and determine the final state of execution as a function of the outcome of the corresponding experiments and trials respectively. They are also used to limit the execution time of tests with bounds on the maximum runtime or the maximum number of failures.
Workload - The workload entity (Fig. 5.8) allows the user to specify the different named sets of operations to be executed against the SUT during the performance tests. The user can specify multiple named sets of operations, representing different utilization scenarios of the SUT that have to be executed in parallel. Within a given named set, operations can be split into multiple subsets relying on *Workload Items*. An example for an e-commerce SUT could be a set of operations named "clients" (currently mainly HTTP requests to the SUT) simulating clients browsing the catalog (first workload item) and buying goods (second workload item), and a set of operations named "admins" of website admins adding new items to the catalog, represented in a single and

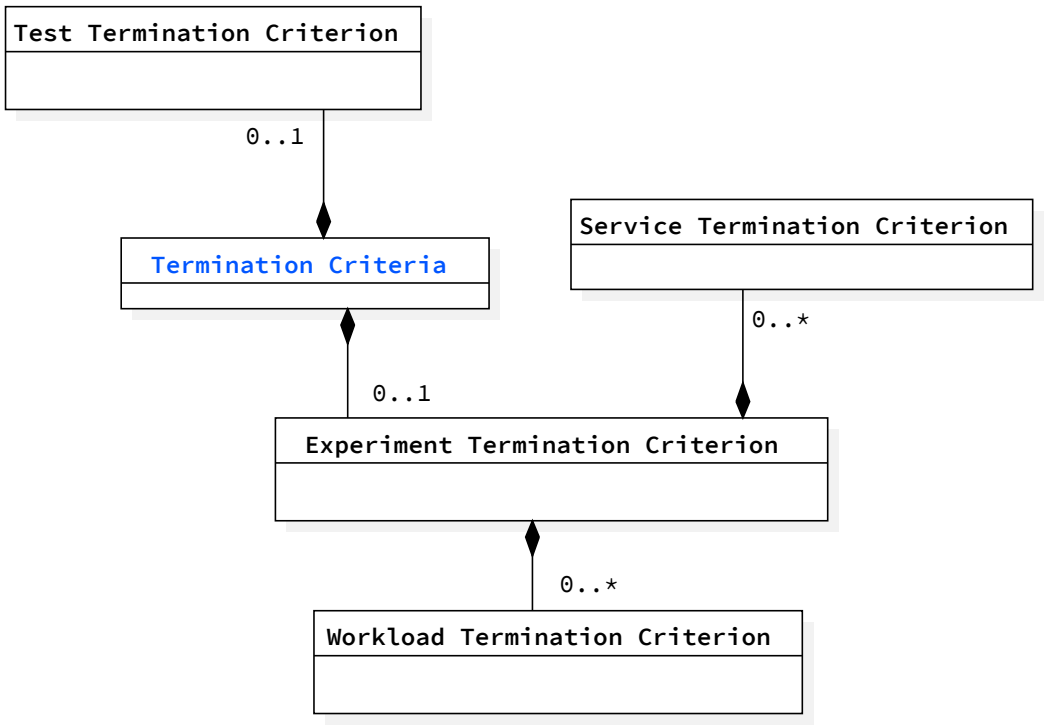

Figure 5.6. DSL: The Termination Criterion Meta-Model

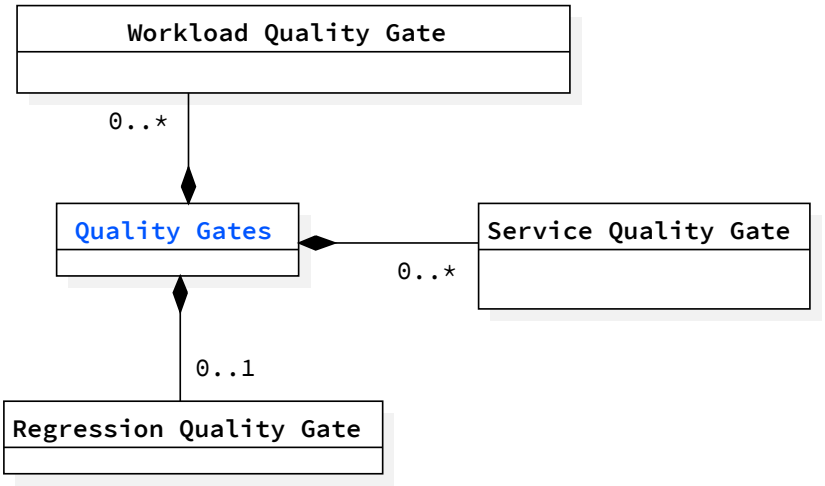

Figure 5.7. DSL: The Quality Gate Meta-Model

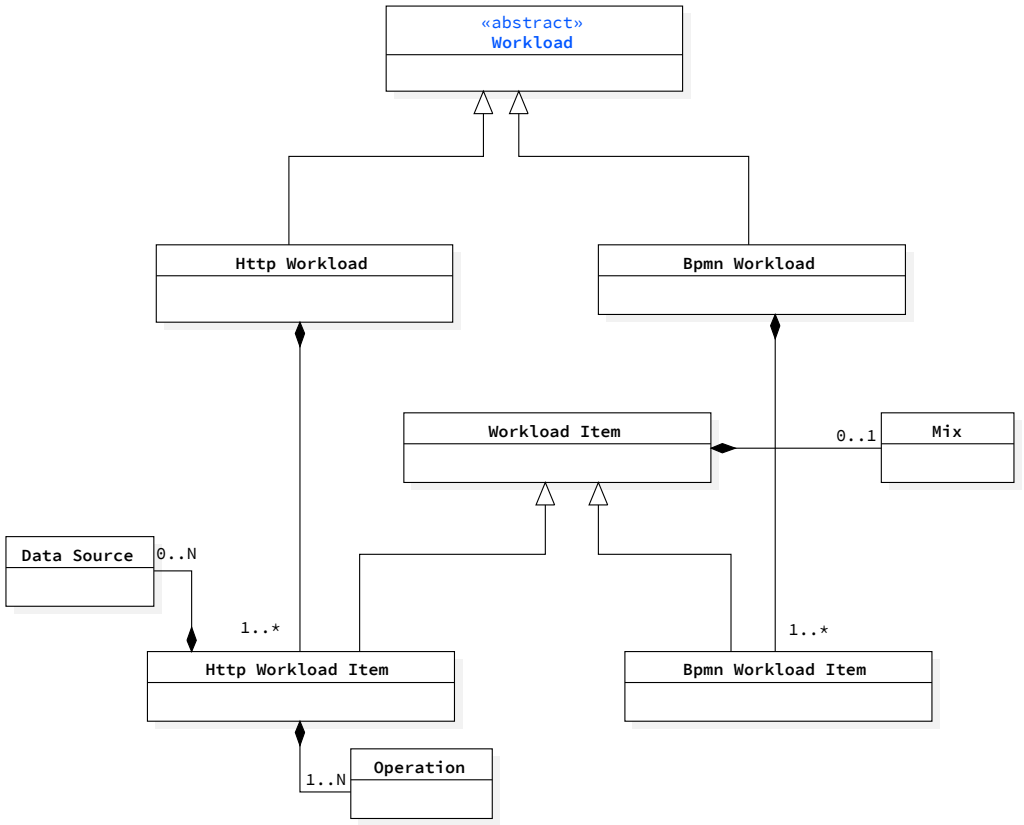

Figure 5.8. DSL: The Workload Meta-Model

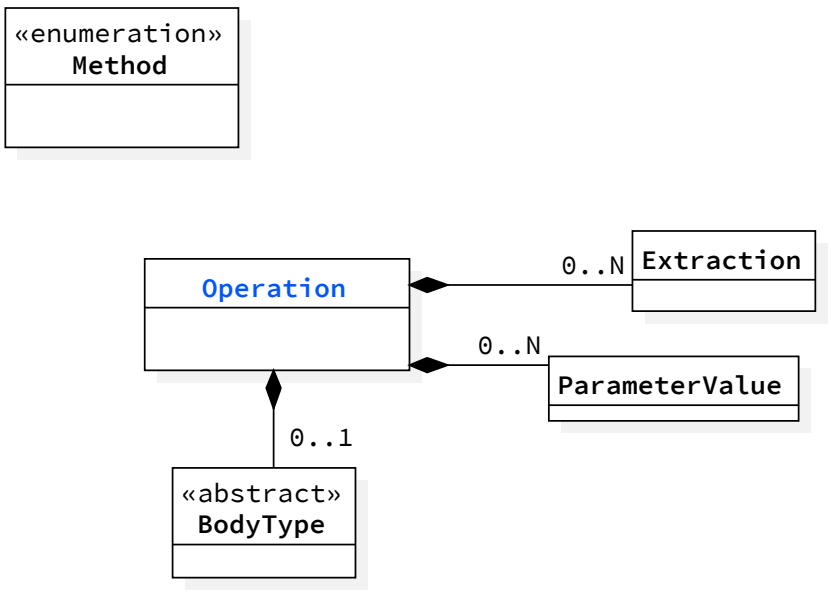

Figure 5.9. DSL: The Operation Meta-Model

dedicated workload item. The workload entity implements SUT-awareness, thus offers different types of workload specifications. Currently the DSL supports specifications for WfMSs and HTTP. In defining the operations, the user can also refer to a *Data Source*, such as a file, and can define the behavior of the interaction with the SUT by controlling the timing between operations.

Operation - The operation entity (Fig. 5.9) enables the specification of all the details for HTTP operations. Different HTTP methods are supported, some of which require a *Body* to be specified. The operations accept different parameters, that can be directly provided or extracted from a given list or the response of a previous operation call.

Mix - The mix entity (Fig. 5.10) allows to specify how to mix the different operations part off the workload item. Operations can be mixed in different ways, for example, they can be executed in sequence, or they can follow a Markov chain [Markov, 1906]. The supported mix types are: *FixedSequenceMix* defining a fixed order over the operations; *FlatMix* randomly deciding the next operation based on the corresponding probabilities; and *FlatSequenceMix* a combination of *FixedSequenceMix* and *FlatMix* allowing the user to specify a random selection of fixed sequences, as well as *MatrixMix* that implements a Markov-chain model and select the next operation based on the current operation and the provided probability. For the *FlatMix*, *FlatSequenceMix*,

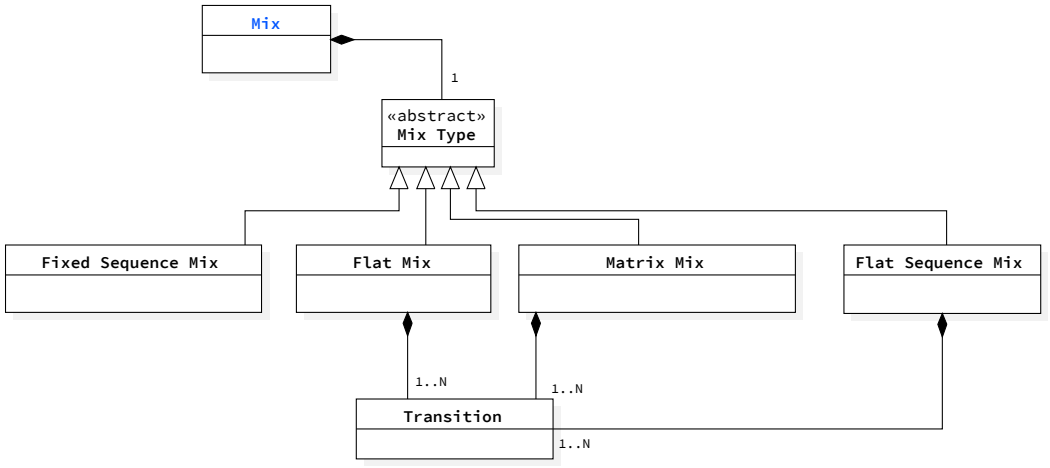

Figure 5.10. DSL: The Mix Meta-Model

and *MatrixMix* types, it is also possible to specify details about the *Transition* among operations.

Sut - The *Sut* entity (Fig. 5.11) allows the user to decide the SUT name and the versions to target as part of the test. It also allows the specification of the default service target of the defined test and its endpoint, and how to determine that the targeted service is ready to accept requests. The user can also optionally specify the configuration of one or more of the services realizing the SUT.

As presented in Fig. 5.12, the version can be specified as a plain string, as a string representing a valid Semantic Version[1], or as a range of valid semantic versions.

Data Collection - To compute the metrics to be observed, data collector services need to be available to collect the raw performance data on which the computed metrics are based on.

We provide the possibility to rely on different collectors, as presented in Fig. 5.13. On the client-side we rely on Faban or JMeter to collect workload performance metrics, thus a Faban or a JMeter collector service can be specified, and optionally configured. If the Faban or the JMeter collector is not specified, but it is required to collect the raw data declared using the *Observe* specification, it is automatically added before executing the test. On the server-side, we provide data collector services for server and service resource utilization, service logs, data on the file system produced by the services, and

[1] https://semver.org/, last visited on February 7, 2021

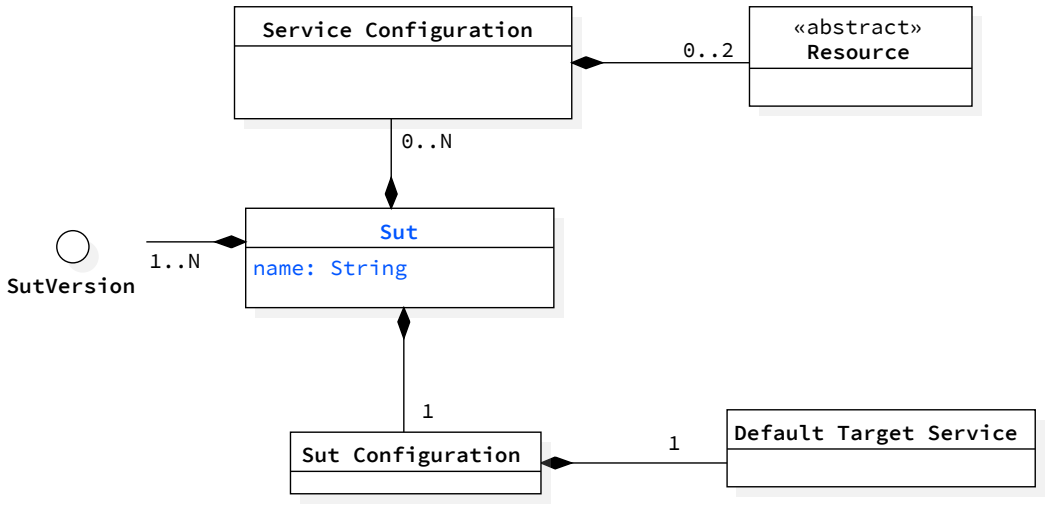

Figure 5.11. DSL: The Sut Meta-Model

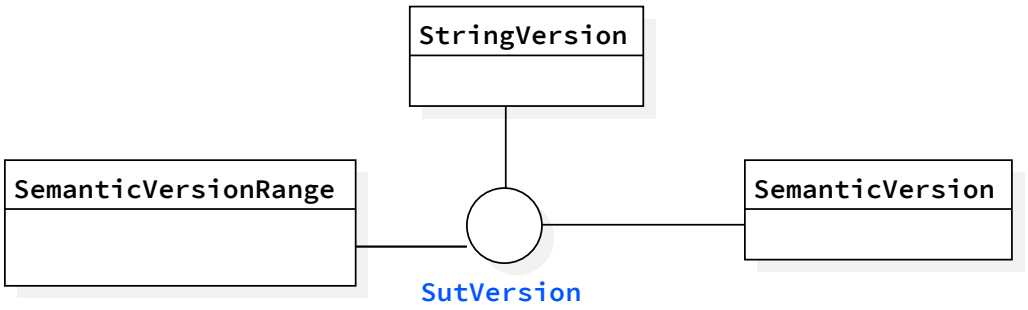

Figure 5.12. DSL: The Version Meta-Model

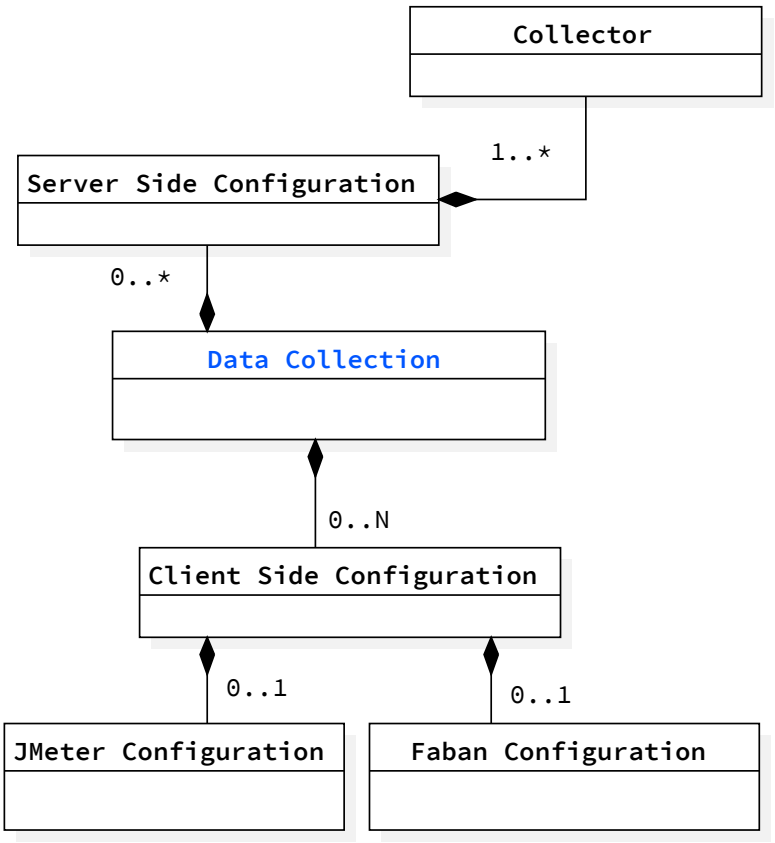

Figure 5.13. DSL: The Data Collection Meta-Model

data stored in databases. The list can be easily extended by integrating new data collector services, according to the user's needs.
The DSL allows users to specify other settings as well, for example, to control the internal behavior of the system executing tests, for advanced users that might want to do so.

5.3.2 Experiment Meta-model

The experiment meta-model is realized by different entities, as represented in Fig. 5.14. The meta-model represents the main entities realizing an experiment and the relations among them. We omit other details because the referred entities are mostly the same as the ones defined for the test meta-model. The experiment meta-model is instantiated from the test meta-model for each experiment that needs to be executed according to the test's goal. As evident from Fig. 5.14, the experiment model misses the goal, and consequently, the experiment does not define an exploration space. For this reason, the main differences between the test and the experiment model are the cardinality of some of the relations (e.g., data collection is always specified even if at test level is not, and there is at least a time-based termination criteria), and the optionality of the fields, e.g., for the users' field in the *Load Function* entity, because for the experiment everything must be precomputed except the number of trials that can be dynamically determined during experiment execution. The version is also different because at the experiment level we released more versions over time compared to the ones released for the test model.

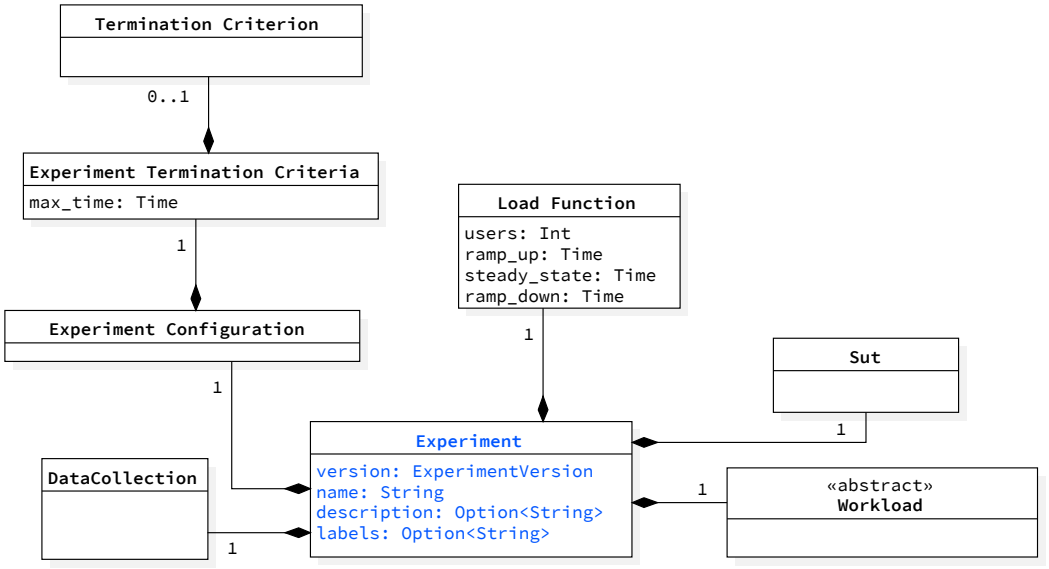

Figure 5.14. DSL: The Experiment Meta-Model

5.3.3 SUT Deployment Descriptor Meta-Model

The SUT deployment descriptor (Fig. 5.15) defines the services part of the SUT, their configuration, and data *volumes* if any, deployment details, and how they communicate among them (i.e., which *networks* the services use). We refer to the Docker Compose [Docker Inc., 2015] standard for specifying SUTs deployment descriptor.

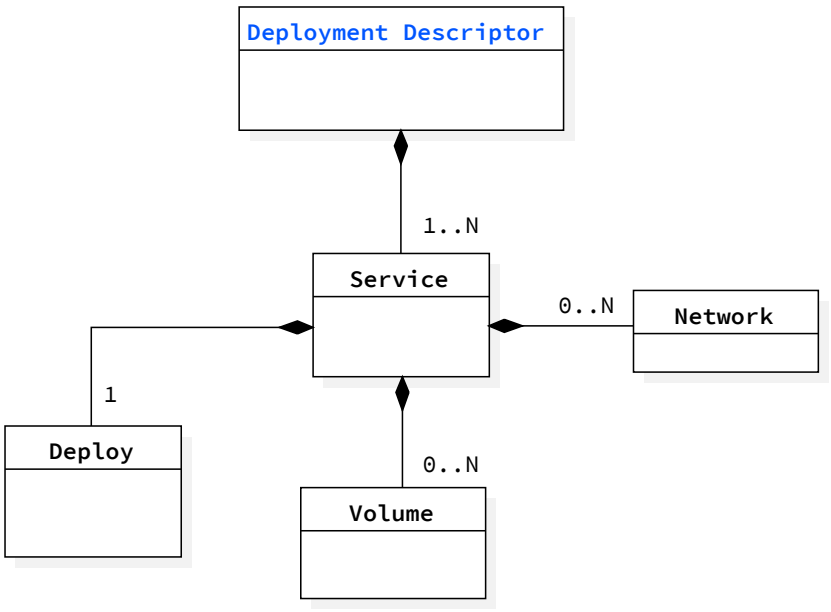

Figure 5.15. DSL: The Deployment Descriptor Meta-Model

5.4 BenchFlow DSL Model

The DSL model provides all the details about the entities and their properties. Due to its complexity, we omit to represent some of the relations among the entities in some cases. The meta-models represent all the relations among entities.

5.4.1 Test Model

Test - In Fig. 5.16 we present the *Test* model. Among the other data, such as the test specification version, the test *name*, *description*, the user can also specify *labels*. Labels are important for selecting different test specifications when integrating test execution in CSDL. Related to the *test* we find the SUT specification, where the main relevant attributes the users specify are the SUT's name and type. Along with the SUT the model supports the *Workloads* specification describing the interactions with the SUT itself. The test configuration supports the specification of the *Goal*'s type and states whether or not reusing previous executions results. Reusing stored knowledge enables the possibility to reuse previous results of tests and experiments defining the same specification, executed as part of previous test execution, or as part of other tests

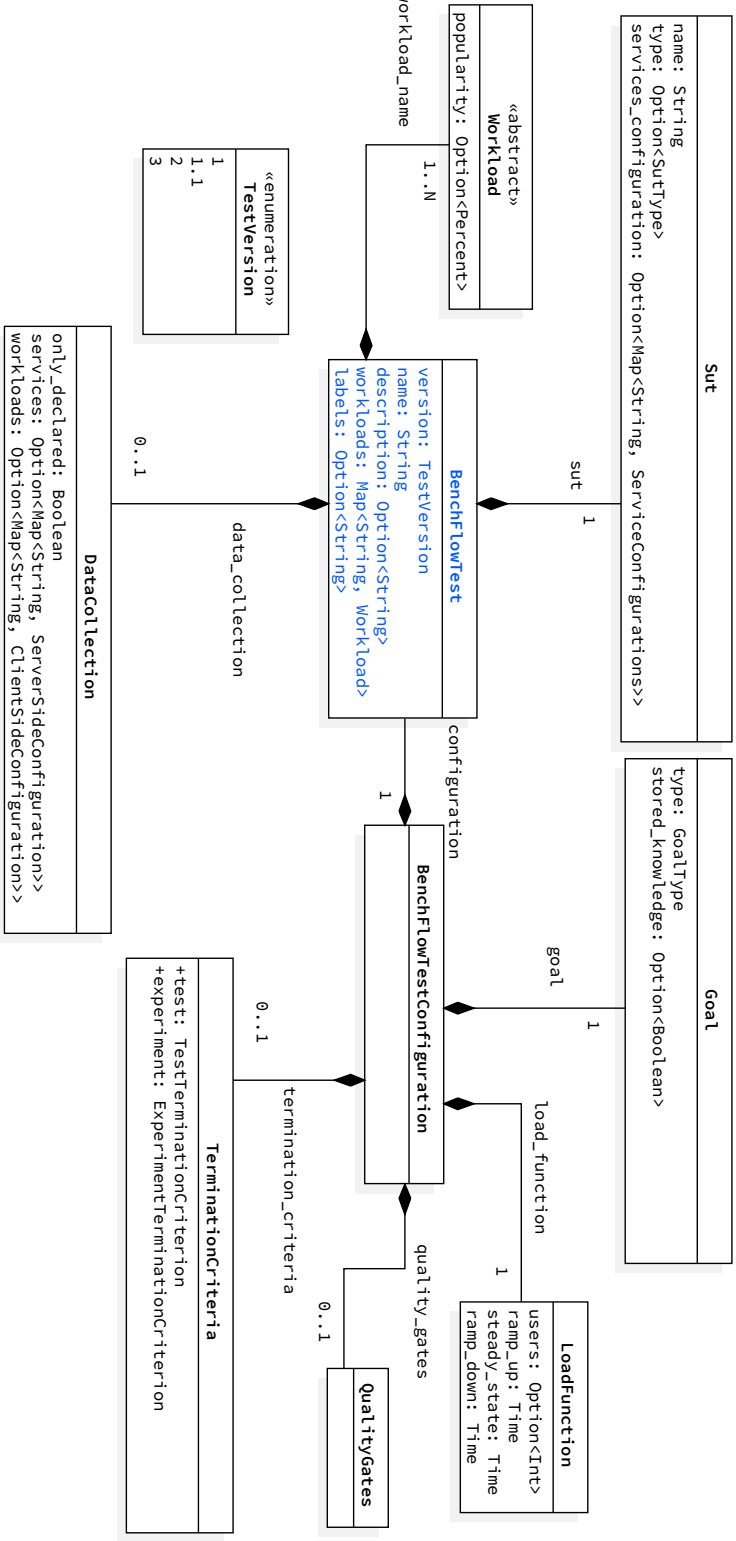

Figure 5.16. DSL: The Test Model

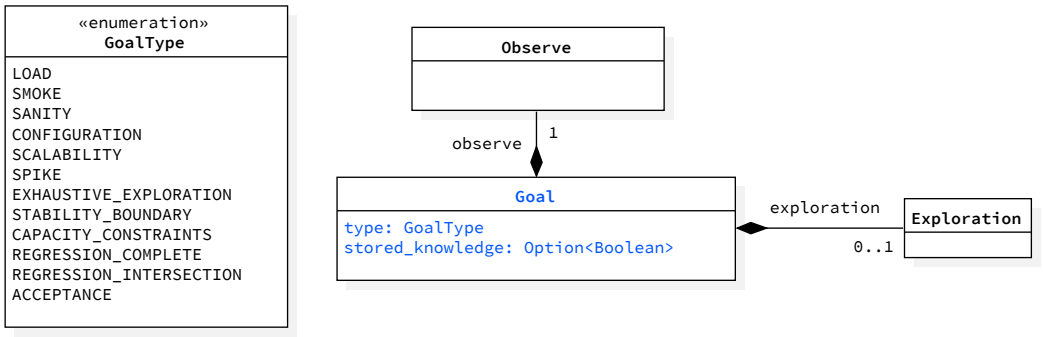

Figure 5.17. DSL: The Goal Model

specifications scheduled in parallel to the submitted one. For what concerns the *Load Function*, the user can specify the wanted number of simulated users and characteristics of the load function in terms of time to go from zero to the number of specified users (i.e., the *ramp-up* time), time to issue the load to the system under test at the specified number of users (i.e., the *steady-state* time), and time to go from the wanted number of users back to zero (i.e., the *ramp-down* time). We currently only support the mentioned shape of the load function, due to limitations in the frameworks we use to automate the performance test execution. Other frameworks can be integrated, thus the shapes of supported load functions could be enhanced. We define a custom type for time, to allow the user more flexibility when specifying time-related values. We support *micro seconds* (`micros`), *nano seconds* (`ns`), *milli seconds* (`ms`), *seconds* (`s`), *minutes* (`m`) and *hours* (`h`). Having specific types allows the system to assert the correct syntax, leading to a failure if it is not correct. As part of the test configuration, the user can also specify *Quality Gates*, *Termination Criteria* for tests and experiments, as well as select and configure the services to collect performance data for SUT's services and the workload.
In the following, we present additional details on the main entities of the DSL, and how the users can specify the different properties of the test definition.
Goal - In Fig. 5.17 we present the *Goal* model. The user can specify many different supported goals. The goal is usually mapped to well-known performance test types. The DSL currently supports the following goals:

1) *LOAD*: representing a load test;

2) *SMOKE*: representing a smoke test;

3) *SANITY*: representing a sanity test;

4) *CONFIGURATION*: representing a configuration test;

5) *SCALABILITY*: representing a scalability test;

6) *SPIKE*: representing a spike test;

7) *EXHAUSTIVE_EXPLORATION*: representing an exploratory test, expecting to explore the entire performance test the user defines;

8) *STABILITY_BOUNDARY*: representing a stability test having the objective of identifying the boundary within the SUT is expected to be stable according to defined quality criteria;

9) *CAPACITY_CONSTRAINTS*: representing a capacity test, where the performance space in which to test for the capacity of the system has well-defined boundaries;

10) *REGRESSION*: represents a regression test among two or more versions of the SUT. Two variants are supported: (a) *REGRESSION_COMPLETE*, where each version of the SUT is tested with the complete workload specified by the user, independently of the operations supported by other versions; (b) *REGRESSION_INTERSECTION*, where all the versions of the SUT are tested with the workload containing only the operations that are present for all the versions.. When more than two versions are part of the same test, the regression is verified between all the possible couples of version numbers.

11) *ACCEPTANCE*: representing an acceptance test based on one or more acceptance criteria.

At least one goal must be specified by the user. The number of supported goals can be extended over time.

Exploration Space - In Fig. 5.18 we present the *Exploration Space* model. The exploration space is required by some of the supported goals, such as the configuration test. When a complex test is specified, the user can define the exploration space in terms of *services*' *resources* and *configuration* (i.e., a map of key-value pairs), and the *Load Function*. Services refer to services defined in the system under test deployment descriptor. For what concerns services' resources, the user can configure CPU and RAM. We define a custom type for CPU and RAM. We support *Millicores* and *Bytes* respectively. Millicores represents a CPU core split into 1000 units (milli = 1000) and is used instead

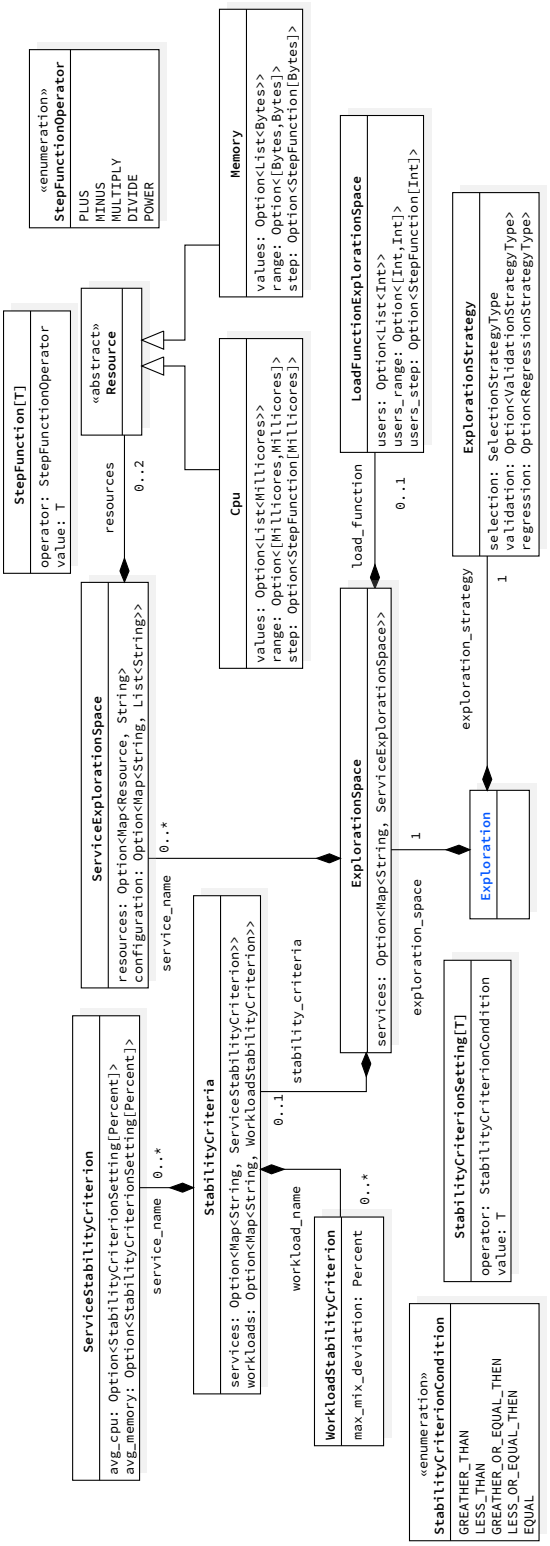

Figure 5.18. DSL: The Exploration Model

of the plain *core* to assign resources to services in a more granular way without the need of using fractions. Bytes can be represented in *kibi* (`Ki`), *mebi* (`Mi`), and *gibi* (`Gi`) [ISO Central Secretary, 2009]. The user can directly specify the values to set over different experiments or specify *range*s to navigate with given *step* functions, that can, for example, apply addition, subtraction, multiplier, division, and power for a numeric variable. The step function determines the step between the values to calculate. When no *step* is defined then the default step of +1 is used to compute all the values of the given variable. The first and last values in the *range* are always included for completeness even though they might not be part of the step function calculation. The exploration space can also be defined on service-specific configuration options. To define a configuration space including a configuration option, the user can specify a configuration option name, along with a list of possible values (i.e., a *variable-values* pair). Configuration options are injected to the SUT using OS environment variables. The exploration of the load function can be defined in terms of the number of simulated users. For some goals, specific data have to be specified, such as for the *STABILITY_BOUNDARY* goal. In such cases, specific entities are provided as part of the exploration space. For the stability boundary test, the *StabilityCriteria* entity is provided, allowing the user to specify stability criteria on services, in terms of CPU and RAM, and workloads, in terms of the maximum allowed deviation of the actual workload issued to the SUT compared to the one defined by the user. The *ServiceStabilityCriterion* is defined using a condition, that can be *greater than*, *less than*, *greater or equal than*, *less or equal than*, *equal*, as well as a given value. To ease the specification of percentage related criteria, we defined a custom *Percent* (`` `%' ``) type representing a percentage value. Our approach computes the entire performance space to explore calculating the Cartesian product without repetitions of all the possible values specified by the user in the configuration space, considering each specified dimension as a vector. For example, if the user defines an exploration space with a *"service_a"* having as resources value for memory: $[128Mi, 256Mi]$ and a load function defining 100 as the number of users, then the exploration space is comprised of two experiments, one with 100 users and $128Mi$ of memory for *"service_a"*, and a second one of with 100 users and $256Mi$ of memory for *"service_a"*. Given the exploration space could explode in terms of experiments to be executed, we provide different strategies to explore the space.

Exploration Strategy - In Fig. 5.19 we present the *Exploration Strategy* model. The exploration strategy allows the specification of three elements: *selection*, *validation* and *regression*, as presented in Fig. 5.19.

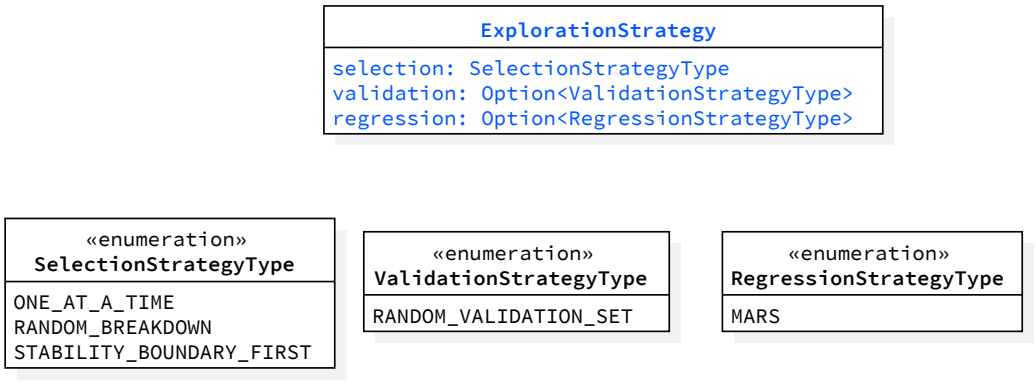

Figure 5.19. DSL: The Exploration Strategy Model

The selection strategy defines the order the experiments in the exploration space are selected for execution. We currently support three *SelectionStrategyTypes*: *ONE_AT_A_TIME*, *RANDOM_BREAKDOWN* and *STABILITY_BOUNDARY_FIRST*. As the name suggests, *ONE_AT_A_TIME* selects the points in the exploration space one after the other. The order is determined by the order in which the exploration space dimensions are defined in the *ExplorationSpace*. The *RANDOM_BREAKDOWN* selection strategy randomly selects the next point out of the ones not yet executed. The *STABILITY_BOUNDARY_FIRST* selection strategy is often used in combination with the *STABILITY_BOUNDARY* goal and uses a binary search to trace the stability boundary. The order of the experiments is determined such that the first to be executed are the ones were the SUT is expected to be less stable, and then applies a binary search in the exploration space to trace the stability boundary, if the SUT is not stable in the first set of mentioned explored points. The current assumption is that the SUT is expected to be less stable were allocated resources to the service are less, and configuration values are expected to be worst. Here the assumption towards the user for the *STABILITY_BOUNDARY* goal, is that the values of a configuration variable are provided in the order from expected worst to expected best performance (e.g., if the SUT is expected to have better performance with more RAM, the specification of a possible value for memory would be: $[128Mi, 256Mi]$). The idea is that in this way the user can start to collect data about regions of the exploration space were the SUT is more likely to be not stable, and although the exploration space could be explored completely in case the SUT is stable in the entire exploration space, she could set termination criteria based on time or on a maximum number of failed experiments to decide when to stop the

exploration. Other strategies can rely on a statistical sampling approach to reduce the number of experiments required to observe the performance over a representative subset of the exploration space [Westermann, 2014]. In such a case, two additional elements are provided: validation and regression. The *validation* strategy is used to select experiments from the *ExplorationSpace* to be used as validation set when determining if the given test has reached its goal or not, e.g., to check if the prediction model provides sufficient accuracy. We currently support a single *ValidationStrategyType*, namely the *RANDOM_VALIDATION_SET*. As the name states, the validation strategy randomly picks points in the exploration space to be used as experiments to be included in the validation set. The *regression* strategy allows the user to select the prediction model to be used to model the performance of the SUT in the exploration space. The built model is used for predicting the result, in terms of completeness/successful/failing execution and observed metrics, of experiments not yet executed to reduce the overall execution time. We currently support a single *RegressionStrategyType*, namely the *MARS*. *MARS* refers to the Multivariate Adaptive Regression Spline (MARS) inference model. The MARS model builds inference models for each of the dimensions to be predicted, named basis functions, and does not require prior assumption as to the form of the data. The MARS model expects at least two metrics to be observed. We build as many MARS models as the number of observed metrics (N). For each model, we use $N-1$ observed metrics as predictors and one observed metric as the predicted one. The validation set is used to compute the prediction accuracy of all the built models, and to determine which of the models requires more data points to be explored so that it can reach the wanted precision. The MARS model automatically includes and excludes terms during the modeling process, meaning it essentially performs automated feature selection. We decided to implement the MARS model as the only supported prediction model, for the time being, because compared to other prediction models it is: fast in predicting new data points, automatically removes outliers from the model, and is precise enough for performance modeling, other than allowing modeling of multiple observe metrics [Ben-Ari and Steinberg, 2007; Van Gelder et al., 2014].

Observe - In Fig. 5.20, we report the specification of the performance metrics of interest for the performance test. The user can request to observe a wide range of performance metrics and statistics provided by the proposed approach and computed on collected performance data, by referring to them by their names. Performance metrics can be observed on the workloads and operations, representing client-side performance metrics, as well as on the services, repre-

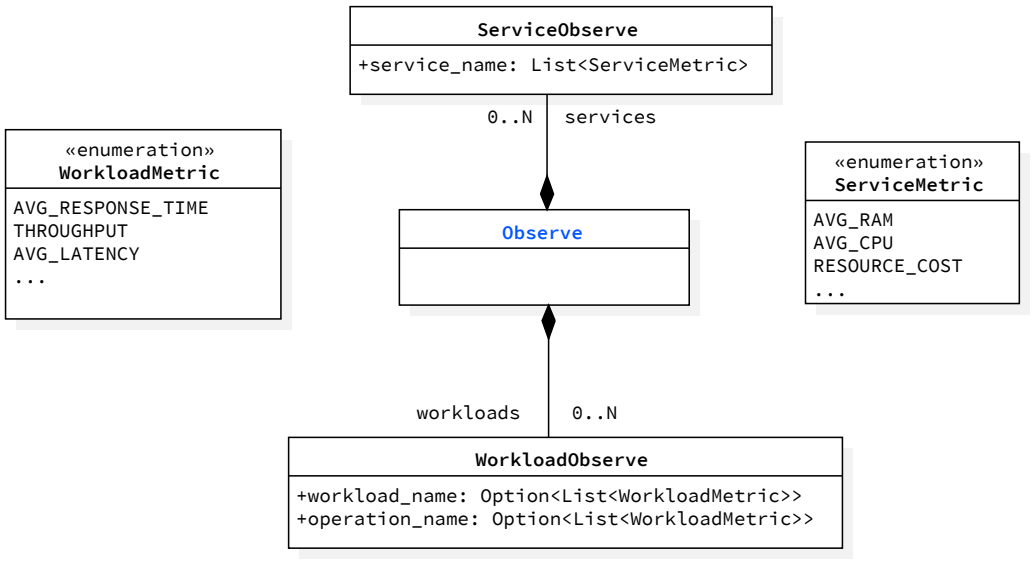

Figure 5.20. DSL: The Observe Model

senting server-side performance metrics. Examples of workload and operation metrics are average response time (*AVG_RESPONSE_TIME*) and average latency (*AVG_LATENCY*) of the SUT under the specified load. Examples of service metrics are average CPU (*AVG_CPU*) utilization and maximum memory (*MAX_RAM*) utilization. We also provide a simple cost-based metric (e.g., *RESOURCE_COST*), assigning an economic value to the resources (*CPU*, *RAM*, *DISK*, and *NET*), utilized by the SUT to sustain the issued load. We also provide SUT-specific metrics, for example for BPMN 2.0 WfMSs we provide custom metrics about the number and time of executed process instances, as well as other ones [Ferme et al., 2017b]. On all the metrics we always make available descriptive statistics, and statistical tests to check for the homogeneity of the collected data (e.g., coefficient of variation[2], and Levene's test[3]), that is for example useful to validate whether the collected data over multiple trials of the same experiment exposes the same behavior.

Termination Criteria - In the Fig. 5.21, we report the model behind the specification of the criteria used to determine if the test execution has to be terminated before its completion. Given performance tests may need a considerable

[2] http://www.ats.ucla.edu/stat/mult_pkg/faq/general/coefficient_of_variation.htm, last visited on February 7, 2021

[3] http://www.itl.nist.gov/div898/handbook/eda/section3/eda35a.htm, last visited on February 7, 2021

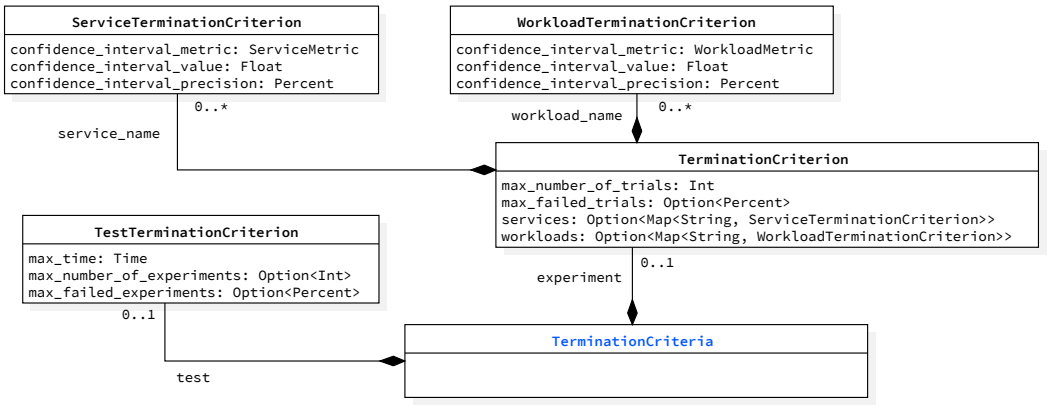

Figure 5.21. DSL: The Termination Criteria Model

amount of time to be executed, the support of tests exploring potentially vast performance space, and the intrinsic fact performance test could fail or have non-deterministic issues during execution (e.g., due to instability of the SUT, the load drivers, or the trial execution framework), it is of utmost importance to specify criteria controlling when to stop tests that are expected not to be successful. A termination criterion can be specified on the test itself, or experiments. At the test level, the users must state the maximum time the test is allowed to be executed (max_time), as well as she can specify the upper bounds for the number of executed experiments ($max\_number\_of\_experiments$) or the number of failed experiments ($max\_failed\_experiments$). An experiment can fail for different reasons, e.g., in case of unexpected errors in the deployment of the SUT or the impossibility of issuing the workload, or due to failures in passing the *Quality Gates*. At the experiment level, the user must specify the maximum number of trials ($max\_number\_of\_trials$), as well as she can specify the maximum number of failing trials (max_failed_trials). A trial represents a repetition of a single experiment. Software systems metrics are expected to vary across multiple experiment iterations, thus it is important to repeat the experiment multiple times, to compute reliable performance metrics. The user can also define termination criteria based on confidence intervals [Boudec, 2011] of workloads and services metrics, specifying the confidence interval metric ($confidence\_interval\_metric$), the confidence interval reference value ($confidence\_interval\_value$), and the precision of the confidence interval ($confidence\_interval\_precision$). The confidence interval is then used to dynamically compute the number of trials. The number of needed trials depends on the type of tested system [Boudec, 2011; Hoefler and Belli,

2015]. Given that the wanted confidence interval can happen to not be reachable, due to the variability of performance measurements, the experiment, in that case, is terminated at the latest when the maximum number of trials is reached. Termination criteria are evaluated during a test and experiment execution. The result is determined by evaluating every single criterion for the given entity and then applying the *OR* operator to the criterion conditions result. As soon as one criterion results verified for the given entity, the execution is terminated. For example, for a time-based test termination criterion, as soon as the criterion is met the test is terminated. For termination criterion based on confidence intervals for experiments, as soon as the criterion results valid, the iteration of the experiment in multiple trials is terminated.

Quality Gates - In the Fig. 5.22, we report the specification of the quality gates. Users specify quality gates to define the outcome of the test according to different criteria, thus contributing to the goal-driven specification of the performance test. The outcome of the test at the end of its execution is usually *successful* or *failing*. The result is determined by evaluating every single gate and then applying the *AND* operator to the gates' results. Quality gates can be specified on workloads and service metrics. The selected *gate_metric* can be compared using a condition, that can be *greater than*, *less than*, *greater or equal than*, *less or equal then*, *equal*, *percentage more*, or *percentage less*, to a given value. For metrics, users can always report both a threshold indicating the wanted value for the metric (*gate_threshold_target*), as well as a minimum accepted value (*gate_threshold_minimum*). The gates are evaluated according to the minimum accepted value. Reporting the expected threshold as well is important for better evaluating the test result. Some quality gates are specified using dedicated metrics, such as *max_mix_deviation*, defining the maximum accepted deviation from the specified workload to interact with the SUT, or *max_think_time_deviation*, defining the maximum accepted deviation in the time interleaving multiple operations interacting with the SUT concerning the specified settings. The maximum allowed deviation is relevant to account for possible errors in the interactions with the SUT because erroneous interactions are not counted as part of the results, thus introducing variations in the specified mix or on the think time.

When relying on an exploration strategy building a prediction model, the user can also specify a quality gate to determine when the test can be considered successful according to the precision reached by the model. We support the Mean Absolute Error (MAE) (*mean_absolute_error*) as model precision metric. If the user does not specify a value for the metric, the default value for the MAE is set to 30%, according to standard performance literature [West-

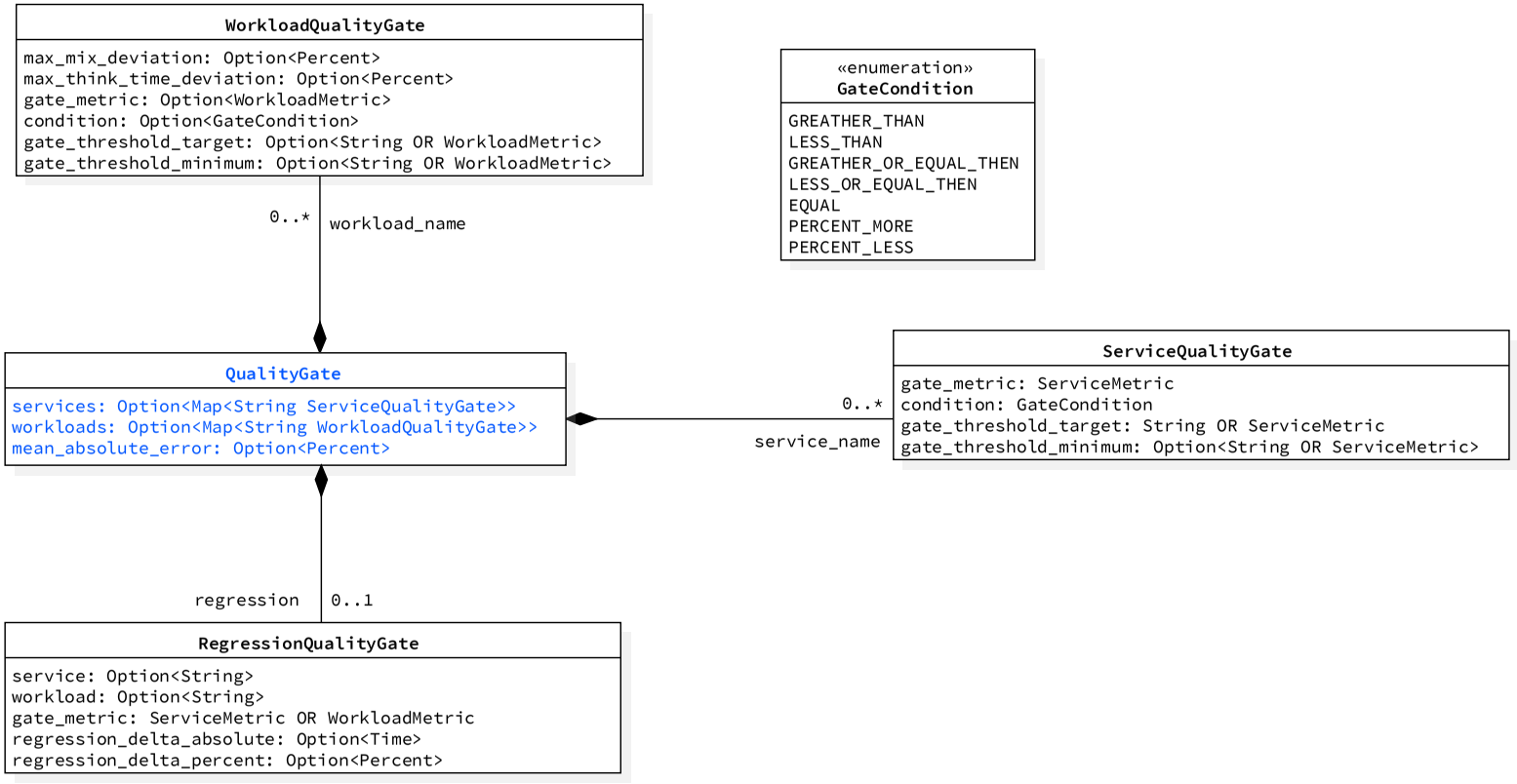

Figure 5.22. DSL: The Quality Gate Model

ermann, 2014]. The validation set is used to compute the prediction accuracy of all the built models, and we expect all the models to respect the prediction criteria expressed by the user as part of the quality gates.
For specific test types, such as the *regression*, the model supports the specification of customized quality gates. In this case, a specific element is provided as part of the model, to facilitate the specification for the user. We currently support a single regression condition based on an observed metric, applied either on a workload or a service. The condition checks for an absolute (*regression_delta_absolute*) or a percentage variation (*regression_delta_percent*) among the versions specified as part of the regression test. The actual sign of the variation triggering a regression depends on the regression *gate_metric*. According to the given metric, a positive or a negative variation in the metric triggers a regression.
Workload - In Fig. 5.23 we present the *Workload* model. We support workload specifications for both RESTful Web services, selecting *HTTP* as *driver_type*, and BPMN 2.0 WfMSs, selecting *START_BPMN* as *driver_type*. The user can specify multiple *workloads*, representing different interactions with the SUT, and their *popularity*. The popularity represents the percentage of requests that should be issued to the SUT from the given named workload. When no popularity of one or more workloads is defined, the overall not assigned popularity summing to 100% is split equally among the workloads not specifying a popularity value. The workload can be realized by different workload items (e.g., representing different scenarios of interaction with the SUT), target a specific SUT version (*sut_version*), and can specify the target service (*target_service*). The workload items are identified by a unique name. When using a Markov chain, the *INITIAL_STATE* name is reserved for identifying the starting state of the chain. According to the *driver_type* the user has different ways to specify the operations, for the *START_BPMN* driver the operations are represented as a list of named BPMN 2.0 process models. It is also possible to specify how to select the timings among the workload items operations (*inter_operation_timings*) and the *popularity* of each workload item, working the same way as the workloads, useful to decide the weight of a given workload when multiple workload items are specified. The *inter_operation_timings* can be one of *NEGATIVE_EXPONENTIAL*, *UNIFORM*, or *FIXED_TIME*. *NEGATIVE_EXPONENTIAL* randomly selects the think time from a negative exponential distribution [Montgomery and Runger, 2013]. This is the timing characteristic simulating the interactions of a large number of users. *UNIFORM* randomly selects the think time from a uniform distribution [Montgomery and Runger, 2013]. *FIXED_TIME* uses a fixed value for the think

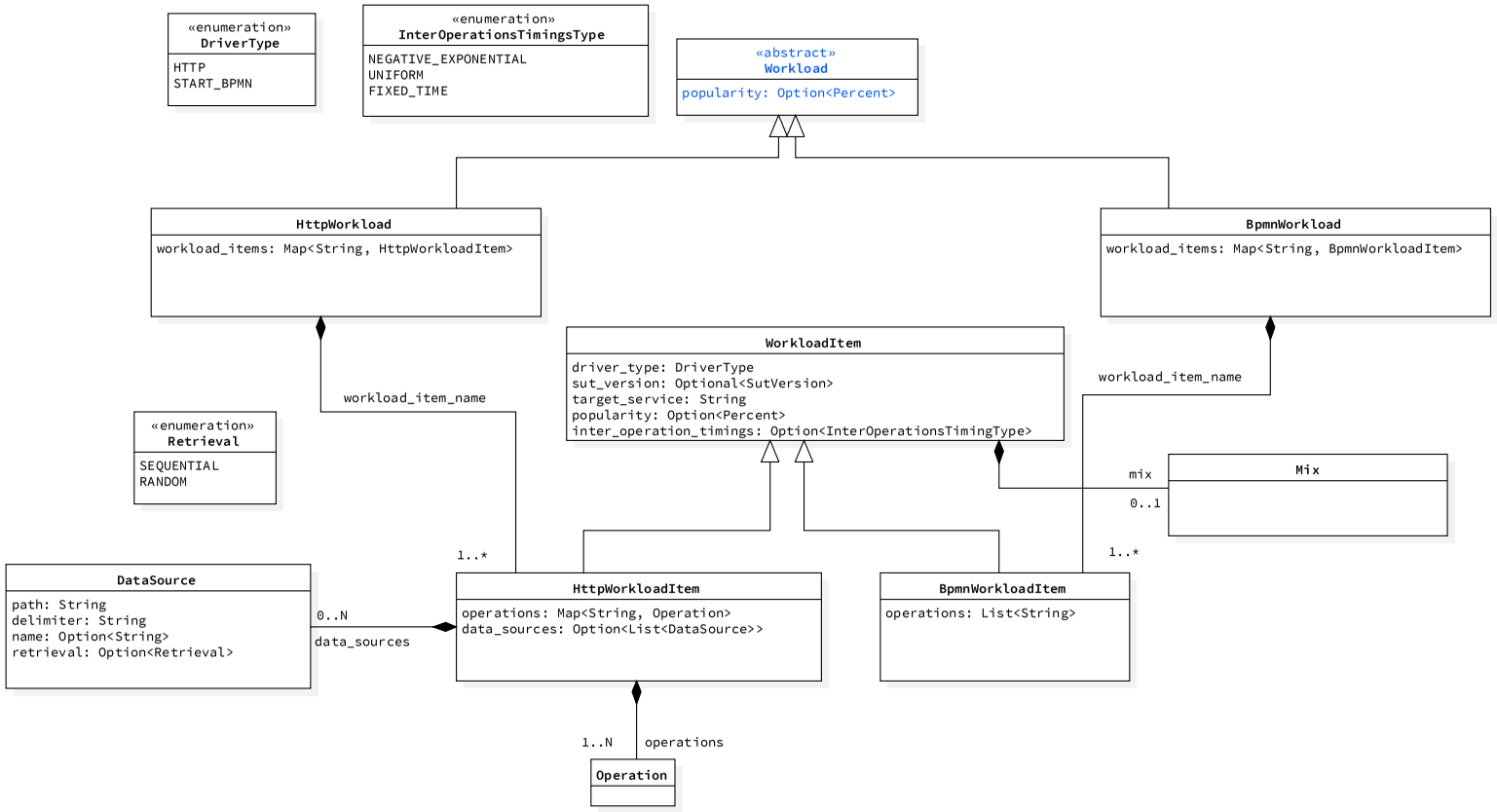

Figure 5.23. DSL: The Workload Model

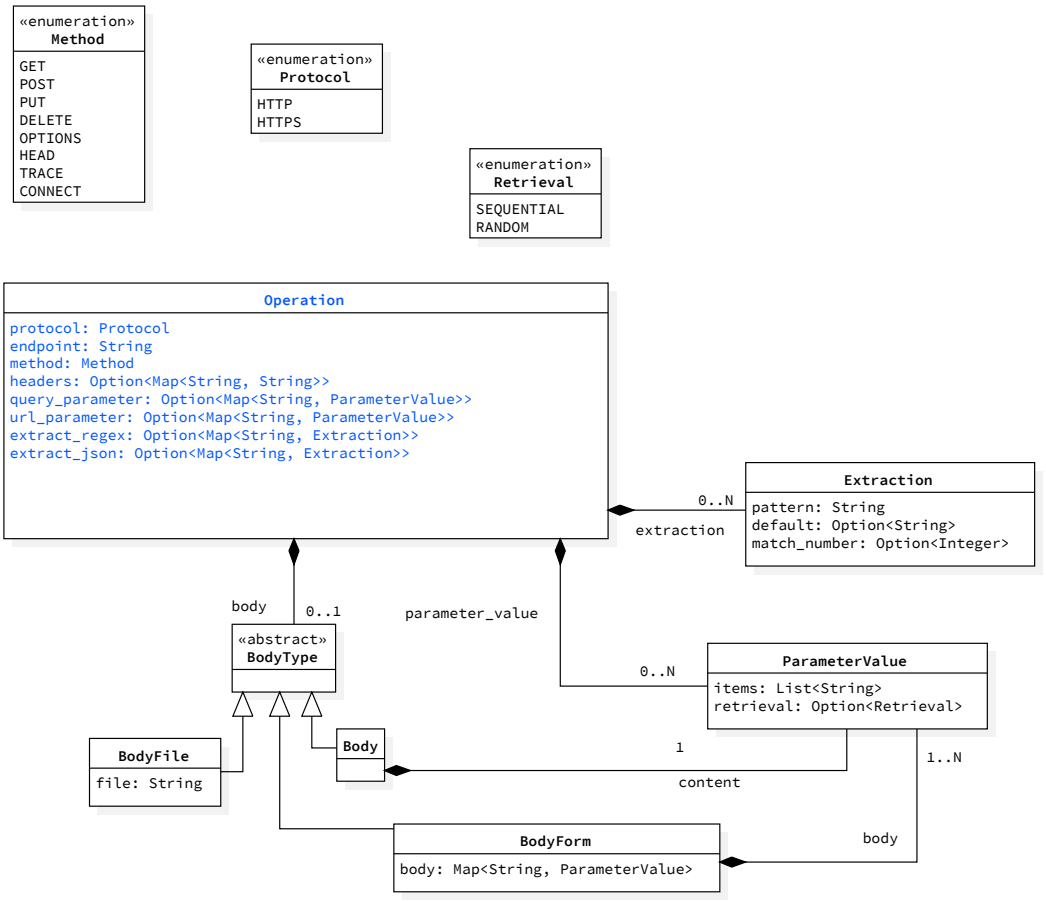

Figure 5.24. DSL: The Operation Model for HTTP

time. Workload items can define *DataSource*s, representing the source to use for the data used by the workload item's operations. Data sources can be files containing data represented using the file *path*, for example, CSV files, and the user can specify a *delimiter* to split the data in the file, as well as assign the file an identifier *name* and define the way to retrieve (*retrieval*) the data. Data retrieval can be *SEQUENTIAL* or *RANDOM*.

Operation - In Fig. 5.24 we present the *Operation* model. As part of the *HTTP* workload, the user specifies *Operations*. Most of the elements of the Operations model for HTTP are related to the RFC standards 7230[4]-7237[5]. An operation must specify the *protocol*, being one of *HTTP* or *HTTPS*, must specify an *endpoint* representing the Uniform Resource Locator (URL) to call,

[4] https://tools.ietf.org/html/rfc7230, last visited on February 7, 2021
[5] https://tools.ietf.org/html/rfc7237, last visited on February 7, 2021

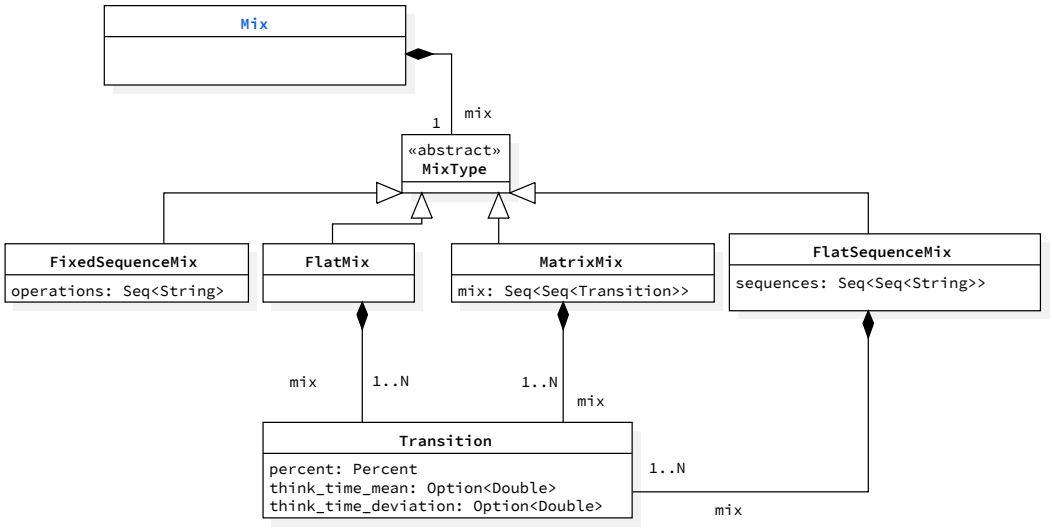

Figure 5.25. DSL: The Mix Model

must specify the *Method* as well, choosing among *GET*, *POST*, *PUT*, *DELETE*, *OPTIONS*, *HEAD*, *TRACE*, or *CONNECT*. A *header* can optionally be provided as well. For the *HTTP* driver type, the operations can extract data from SUT's response to be used in subsequent operations and can refer to *DataSource*, as a source of data for calls to the SUT. *Extraction* of data can happen using a regular expression applied on a plain text or a JavaScript Object Notation (JSON) object. To extract the data, the user has to specify a pattern, and optionally provide a default value in case the patterns does not match any element, as well as a *match_number* to select a specific element in the ordered set of the matched elements, in case more than one matching is found using the regular expression. For some methods, for example, the *POST* one, a body payload is required. The body payload can be either a file, a list of possible body strings, or a tuple of <key><value> pairs for a body form. The value of the tuple is *ParameterValue*s, representing a list of strings selected *SEQUENTIAL*ly or *RANDOM*ly. In addition to the body form, *ParameterValue*s can be used in the query string and in the URL, which are in both cases tuples of <key><value> pairs.

Mix - In Fig. 5.25 we present the *Mix* model. To decide how to mix the operations of a workload, the user specifies the *Mix*. The mix allows the definition of sequences of operations, or Markov chains controlling how to navigate from one operation to another one, as well as the time the simulated user spends to think about the next operation to execute. The user specifies the think

time relying on the *Transition* entity. The *Transition* entity allows the user to specify both the percentage of time the operation is executed in the mix, as well as the think time using a normal distribution specified using the mean (*think_time_mean*) and the deviation (*think_time_deviation*). The framework executing the test attempts to respect as much as possible the specified mix of operations and think time. Due to the indeterminism of software system behavior, a slight deviation from the same must be expected according to the known variance of SUT's behavior, the test execution environment, as well as the SUT deployment environments.

SUT - In Fig. 5.26, we report the model of the SUT and its configuration (*Sut*). The users can specify the *name* of the SUT and the target *versions*, as well as the *type*, being one of *HTTP*, RESTful Web services, or *WFMS*, for BPMN 2.0 WfMSs. As reported in Fig. 5.27, the version can also be specified as a *SemanticVersion* or as a *SemanticVersionRange* other than a *StringVersion*. When specified using the semantic versioning standard, we refer to the version 2.0 of such standard[6] and we rely on the *com.github.zafarkhaja.semver.Version*[7] library.

The SUT has to be configured to automate the test. The user configures the *DefaultTargetService* for the workload, specifying the service *name* and the default *endpoint*, and optionally states how to identify when the SUT is ready to receive the load (*sut_ready_log_check*). The user can also specify *ServiceConfigurations* in terms of service' resources and configuration variables, similar to how the same data can be configured for the *ExplorationSpace*. In this case, a single value must be specified, and no *range*s are accepted. The proposed framework can also take care of deploying the SUT for different experiments, thus the user can specify data about the deployment of different services. For each service, the user can specify the name of a target server where she wants to deploy that service. The name of the servers can be configured as part of the framework part of the proposed approach. Setting custom configurations and deciding which services to deploy on which server, allow the user to have control on the way the services of the SUT has to be started, for example, to rely on stubbing mechanism that might be available in the services, so that to isolate the service from dependent services (e.g., to avoid cyclic dependencies) for the performance test. If no deployment information is provided, the proposed framework selects the best suitable and available servers where to deploy the SUT, unless a full *endpoint* is specified. When a full endpoint is specified, the

[6] https://semver.org/, last visited on February 7, 2021

[7] https://github.com/zafarkhaja/jsemver, last visited on February 7, 2021

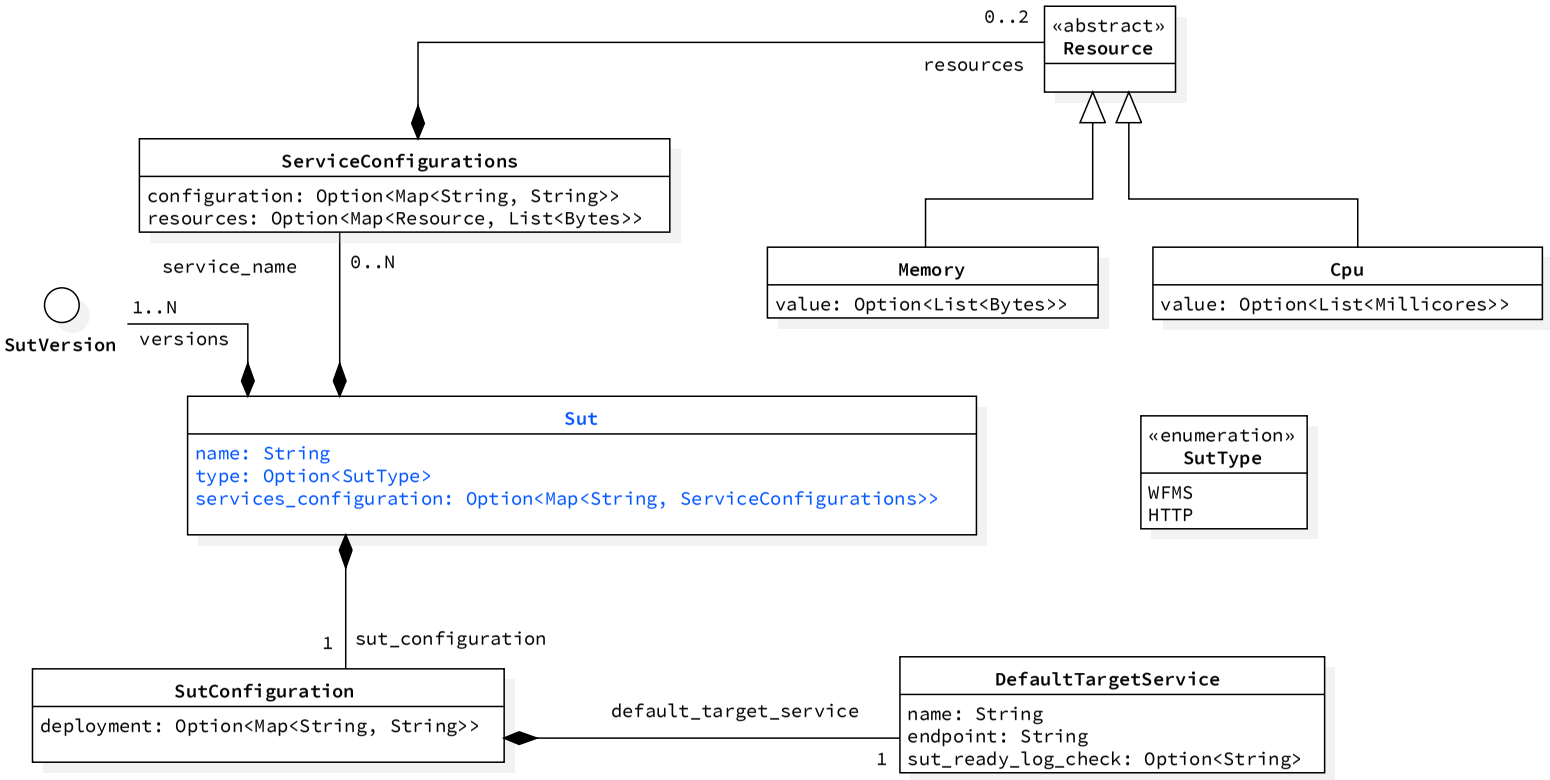

Figure 5.26. DSL: The SUT Model

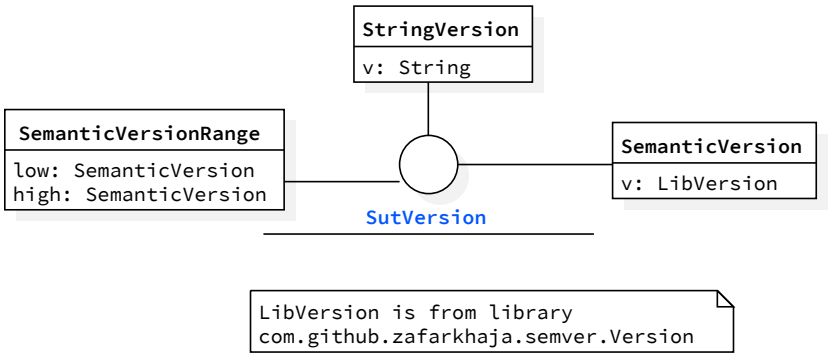

Figure 5.27. DSL: The Version Model

SUT is expected to be already deployed at the specified endpoint, thus the deployment and the service configurations are skipped. It is important to note that for complex tests to be executed, the SUT deployment is expected to be carried out by the proposed framework so that configurations requested for different experiments can be injected at deployment time.

Data Collection - In Fig. 5.28, we report the model of the *DataCollection* services to be used to collect performance data used to compute the observed performance metrics. Data collection services are available as part of the proposed solution and can collect client-side performance data (*ClientSideConfiguration*), and server-side (*ServerSideConfiguration*) resource utilization data, files produced by the SUT as well as dumps of databases (e.g., useful for computing BPMN 2.0 WfMSs-specific metrics). Client-side collection services depend on the actual trial execution framework used to execute the experiments (either *JMeterConfiguration* or *FabanConfiguration*), and the user can specify the collection *interval* between two subsequent samplings of performance data. Server-side collection services are of different types, according to configuration requirements they might have. A user can specify a single collector (*CollectorSingle*), or a list of collectors (*CollectorMultiple*) attached to a service. When a collector has to be configured, *CollectorMultipleConfiguration* is used, and it can be configured according to SUT configurations (e.g., to properly connect to a DBMS) and the list of services can be extended according to new requirements. Values of configuration variables can refer to keywords representing variables injected by the framework part of the proposed approach at runtime, e.g., Internet Protocol (IP) addresses and ports of services from which data have to be collected, or configurations already specified as part of the SUT deployment descriptor part of the test bundle. For many data collector services

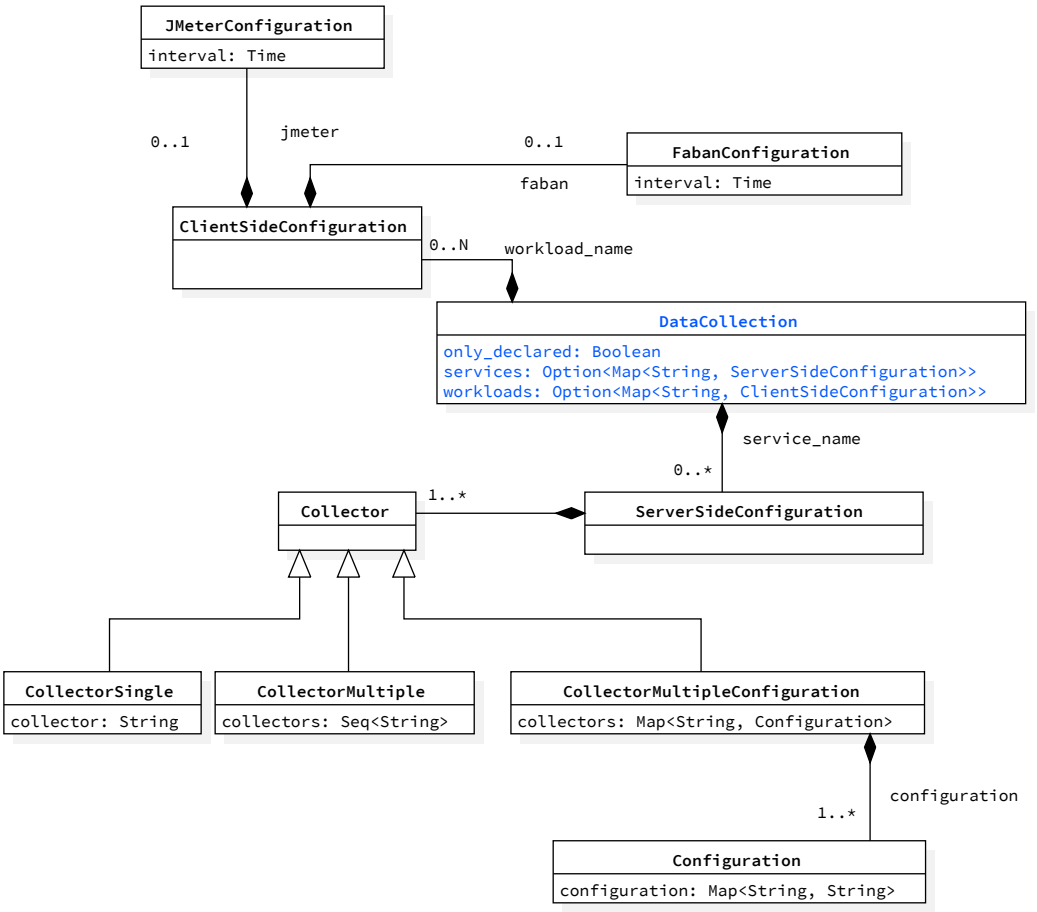

Figure 5.28. DSL: The Data Collection Model

we provide defaults, for others, we require the user to configure the collector so that it can access the data (e.g., the database data collector services require configuration to be able to access the database, and collect the wanted data). The proposed framework takes care of injecting collection services for the observed metrics, in case the user does not specify some of them and no custom configuration is required. To be able to collect all the data required to respond to current and future users' needs, by default all collectors not requiring configurations are enabled on all the services. The user can disable this behavior by setting *only_declared* to `true`.

5.4.2 Experiment Model

In Fig. 5.29, we report the *Experiment* model generated by the framework part of the proposed approach, starting from the *Test* model. The *Experiment* is characterized by a *name*, a *version* for the model used to represent the entities, and an optional *description* and *labels*. Many of the entities are directly derived from the test model, for example, the *Sut* entity, the *DataCollection* entity, and the *Workload* entity. The *BenchFlowExperimentConfiguration* is generated from the data part of the *Test* model, mainly from the *Goal*, *LoadFunction*, and *TerminationCriteria* entities of that model. An experiment always defines a complete *LoadFunction*, in terms of *users*, *ramp_up*, *steady_state*, and *ramp_down* and has at least a *TerminationCriterion* based on time (*max_time*) to guarantee an upper bound in the execution time. The *max_time* is computed starting from the *max_time* termination criterion defined for the test, divided by the number of experiments (and trials) to be executed according to the test specification. For what concern the other *ExperimentTerminationCriteria*, the user must define the *max_number_of_trials*, to control how many times at most the experiment can be repeated to improve the quality of the collected metrics, and can specify the number of *max_failed_trials* to ensure the experiment is stopped if trials are failing. All the termination criteria are directly derived from the ones expressed at the test level, given the user is not expected to specify the experiment, but only tests. Quality gates are not present at the experiment level, because they are all defined and evaluated at the test level, even when they are applied to experiments' results.

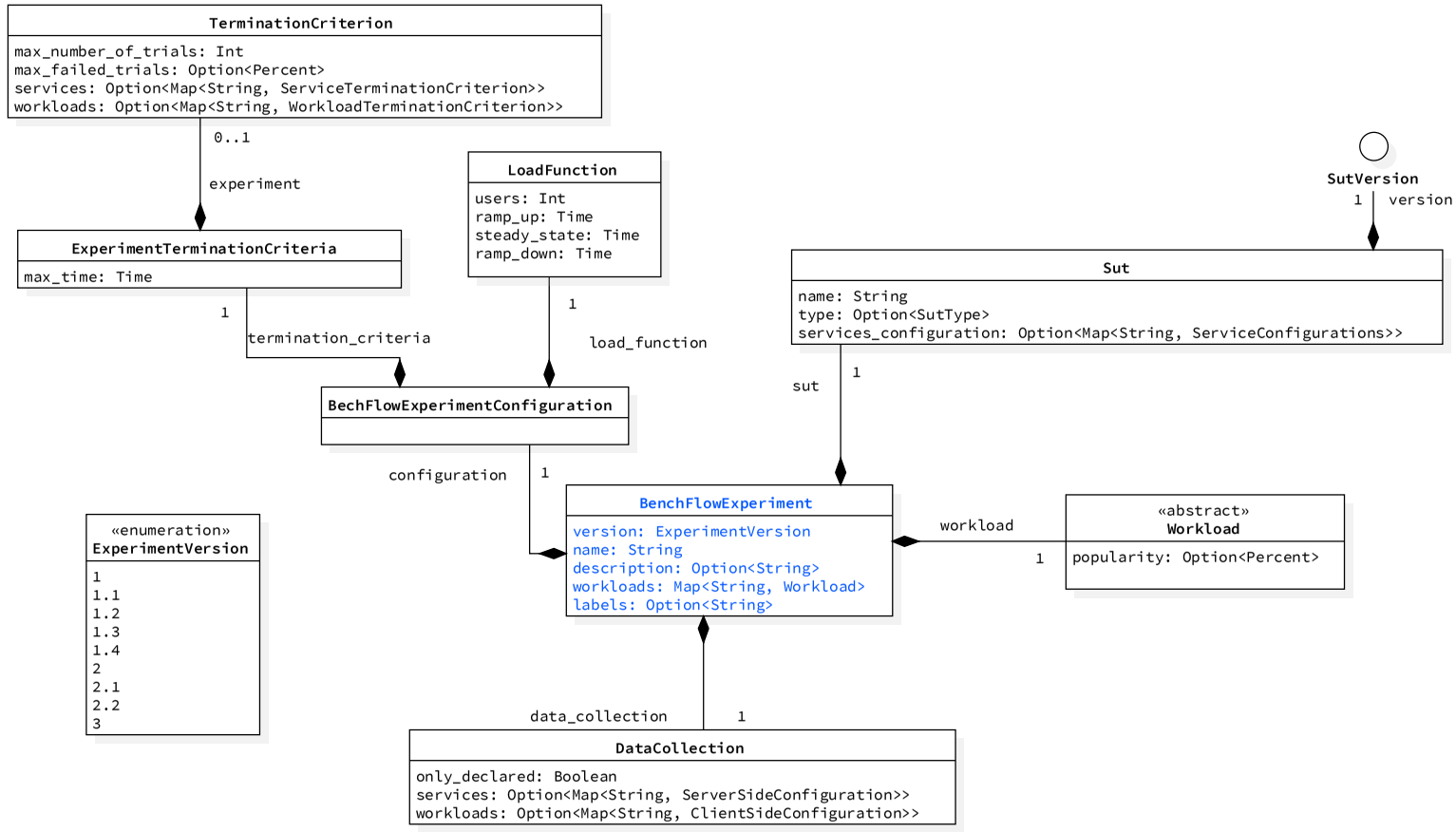

Figure 5.29. DSL: The Experiment Model

5.4.3 SUT Deployment Descriptor Model

As part of the proposed approach, we define SUT deployment descriptors as part of the *test bundle*. We rely on the Docker Compose standard.
In Fig. 5.30 we present the deployment descriptor model, relying on the Docker Compose standard. The model is simplified compared to the actual one available in the referenced standard. We report the relevant entities for the context of this thesis. The deployment descriptor describes all the *Service*s part of the SUT, details on how to *Deploy* each service, data about the used *Volumes*, and the *Networks* the services use to communicate with each other. The services are identified by a *name*, and an *image* representing a reference Docker Image [Docker Inc., 2013] packaging the service the user wants to deploy. The user is also expected to specify data related to how to *deploy* the service, in terms of the number of *replicas*, i.e., how many instances of the given service to deploy, and optionally *placement* constraints, e.g., the name of the servers on which to deploy a given service. The user can also optionally specify a *command* to be executed when a new Docker Container [Docker Inc., 2013] is instantiated out of the Docker Image specification, *environment* variables configuration, *ports* used by the service, ports *expose*d to access the service, named *volumes* used by the service to store data, *network_mode* identifying the *type* of network to use when named networks are not needed, other services from which the given service inherits data volumes (*volume_from*), as well as specific settings to access the properties of the infrastructure underlying the Docker container (e.g., *pid*). Data *volume*s are identified by a name, as well as *network*s are identified by a name. For the *network*s, the user can also specify a *type*, given multiple types of networks can be utilized[8]. Services sharing the *network*s can access other services by referring to their name, or by relying on injected environment variables containing data about how to access the services, among which the *ip* and the *port* of which to reach the service. *Environment* variables configuration can be directly specified as part of the model or can refer to a dedicated file containing them so that all the services needed to access the same environment variables can refer to such a file. It is important to note that the Docker Compose standard can also be deployed on other infrastructures other than Docker. There are different solutions to convert the Docker Compose specification to other specifications, such as the Kubernetes [Kubernetes, 2014] one. Some capabilities are missing or adapted, but this allows the deployment of services defined using the Docker Compose standard on multiple infrastructures.

[8] https://docs.docker.com/network/, last visited on February 7, 2021

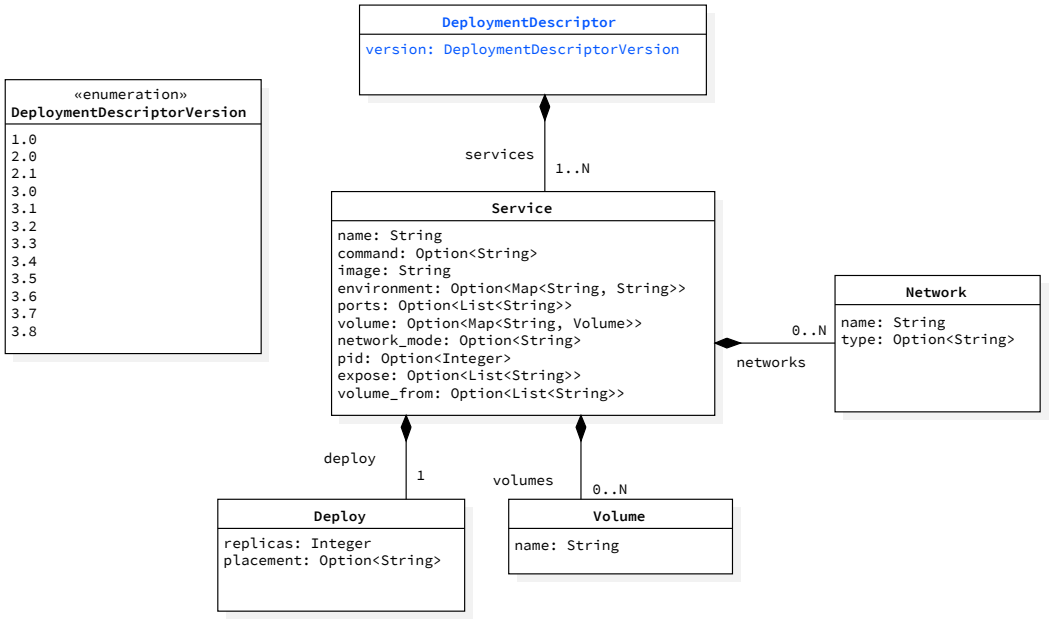

Figure 5.30. DSL: The Deployment Descriptor Model

5.5 BenchFlow DSL for CSDL

The presented DSL allows users to declaratively specify performance tests. When executing those performance tests as part of CSDL users need also to specify additional details, for example, to control when to activate the different performance tests they defined, as well as control boundary conditions and requirements to be in place to maximize the successful execution of the tests. It is important to offer users a way to organize different tests so that then they can decide which one to execute in the different moments of the CSDL. The CSDL extension to the DSL we present in this section has the scope of allowing users to categorize different performance tests, and to describe when and under which condition different set of tests are to be executed.
As presented in Sect. 4.4 code is continuously built by CIS and software is delivered using CDS. CIS and CDS are the right tools where to integrate the CSDL extension to the DSL we present in this section, because we obtain all the data to be used to allow users to configure test selection to be executed, as for example following the indications in Sect. 4.4.3.

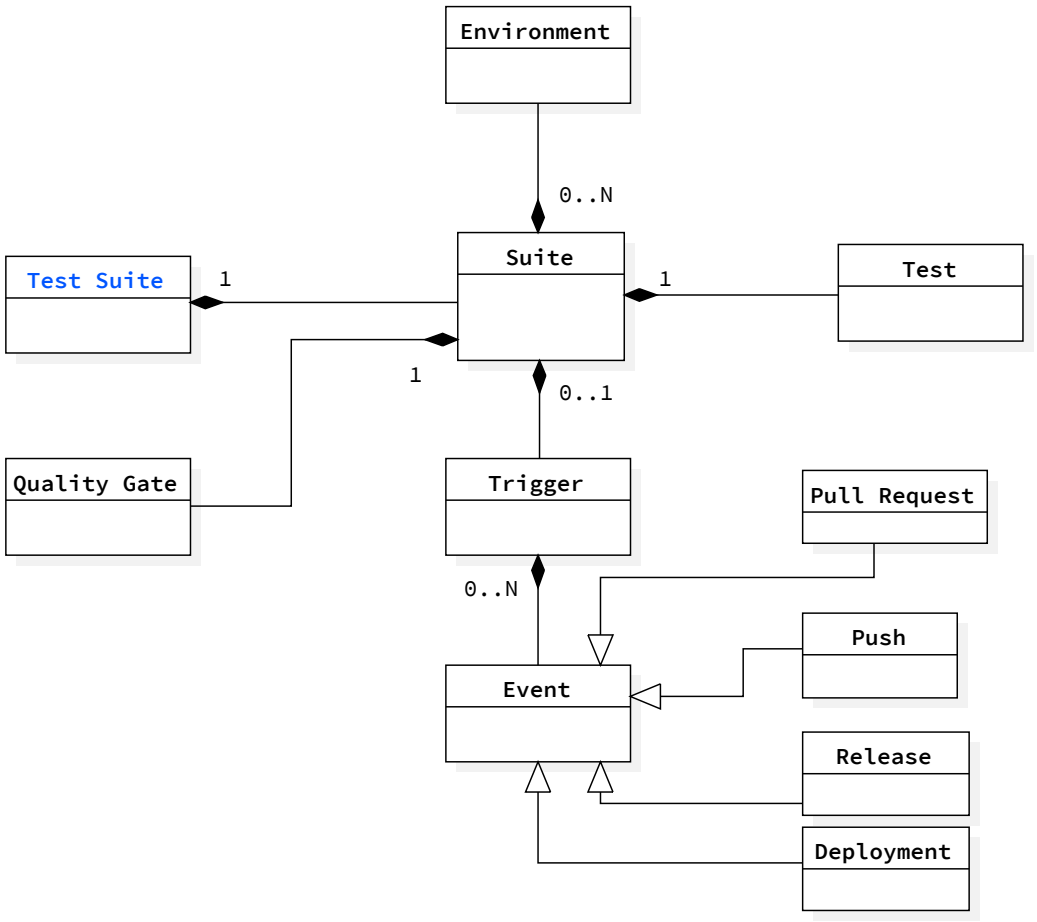

Figure 5.31. DSL: The CSDL Meta-model

5.5.1 BenchFlow DSL for CSDL Meta-model

In Fig. 5.31 we report the specification of the *TestSuite*, identified by a *Version*. The user can specify test *Suite*s, to automatically select one or more tests implemented using the specification presented in Sect. 5.4, in different moments of the CSDL lifecycle according to the generated events. In CSDL software is usually committed to a repository in a VCS by different users, and developed using multiple so-called feature branches. When features are considered mature enough, are usually merged in a development version first and then in a production version of the SUT. These activities generate events such as code *Push*es, *PullRequest*s, *Release*s, and *Deployment*s. These events can be used to determine when to *Trigger* different performance tests. In the case of deployments, the user can also specify the *Environment*s of interest of such deployment to trigger the execution of performance tests. As per the test specification, also for test suites, the user is expected to define a *QualityGate* to determine the overall result of the test suite execution. The quality gates can be applied to the entire suite, or on the test suite excluding a given set of tests. Termination criteria are not required for the integration with CSDL, because each single test part of the test suite is expected to define proper termination criteria.

5.5.2 BenchFlow DSL for CSDL Model

In Fig. 5.32 we detail the specification of the *TestSuite*. The test suite is characterized by a *version*, *name*, as well as, optionally, a *description*. The test suite specifies a *Suite* selecting *Tests* by defining the test to include from the collection of tests defined by the user using the presented DSL. Selection criteria can be based on the *labels* defined on the single defined tests. The user can specify a regular expression matching the labels, and selecting the tests. The selection of the tests to be executed happens in order of file name under a *"./test/performance"* folder part of the same code repository where the SUT is developed. If no tests with the specified labels are found the test suite can not be executed. Alternatively, the user can specify a list of direct *paths* to performance test definitions. The definition of *Triggers* depends on the events based on which triggers the execution of the test suite, and their characteristics. User can specify as triggers:

1) a code *Push*, optionally specifying the *branches* of interest using a regular expression, where the push has to happen to trigger the test suite execution;

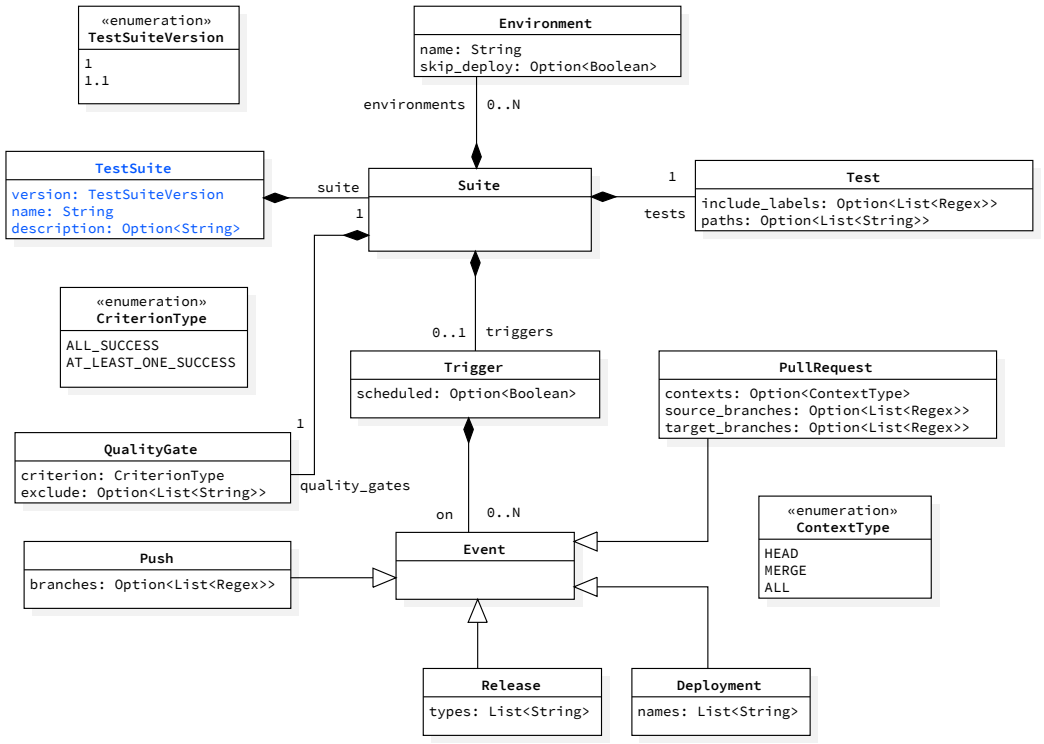

Figure 5.32. DSL: The CSDL Model

2) a *PullRequest*, optionally specifying the *source_branches* and the *target_branches* of the pull request, and the *context* of the code against which executing the test. The context can be *merge*, meaning the test is requested to be executed on the version of the SUT built out of the current source branch merged with the current target branch, *head*, meaning the test is requested to be executed on the version of the SUT as it is in the current source branch or *all* including both of the previous cases. The source and target branches can be selected using a regular expression.

3) a *Release*, identified by indicating the release *types*. The user can indicate an arbitrary list of release types, matching the release types of the CIS triggering the test suite execution.

4) a *Deployment*, identified by an arbitrary list of deployment names, matching the deployment names of the CIS triggering the test suite execution. When a deployment trigger is selected, a named list of deployment *Environments* can be specified, and optionally the user can indicate whether to skip the SUT deployment (*skip_deploy*) or not. If the SUT is already deployed by processes part of the CIS or the CDS systems, the *endpoint* specified in the *sut* part of the model of the executed tests is expected to be specified and executed tests are not expected to require SUT configuration other than the default one.

A test suite can also be *scheduled* for execution, meaning no events trigger the test execution, so that the CIS or the CDS system drive the execution according to the implemented processes (e.g., overnight). When no *Environments* are specified, because not required by the triggers, the tests are going to define how to deploy the SUT and where, and the framework part of the proposed approach is going to handle the provided specification. If the user wants to execute the test only when the SUT is deployed to specific environments, she can always control the deployment and pass the *entrypoint* for the SUT when scheduling the test suite. The *QualityGates* control when the test suite is expected to fail or be successful. The quality gates control the result of the test suites relying on a *CriterionType* applied on all the executed tests, but the optionally *exclude*d ones. User can exclude specific tests from the quality gates evaluation, by enumerating the paths to those test specifications. The test suite can fail if at least one of the quality gates of the tests realizing the test suite fails (*ALL_SUCCESS*), or if none of the tests of the suite is successful (*AT_LEAST_ONE_SUCCESS*). The second option is useful for example when testing different variants of the same SUT and it is sufficient

that at least one test is successful to proceed.

5.6 BenchFlow DSL and DSL for CSDL Implementation

In this section, we provide some implementation details of the proposed DSL. We describe the format we use to represent and serialize the model, YAML, we present some examples of declarative performance test specifications relying on the proposed YAML specification format, and the software library we provide to enable programmatic definition of declarative performance test specifications and parsing and serialization of the model itself. We highlight some of the implementation details and algorithms built to provide the contributed DPE approach for specifying performance tests. We opt for the YAML specification language, because is a widely used human-readable and machine-readable data serialization standard. Additionally, YAML natively include inheritance and facilitate overrides of elements, and is becoming the de facto standard for specifications and configurations in many CSDL tools and framework [IBM Developer, 2019], especially for tools and frameworks supporting declarative configurations, for example, Kubernetes.

5.6.1 DSL YAML Complete Specification

In this section, we illustrate the YAML specification format the users of our approach are expected to master for declaratively specifying performance tests and configuring the test's automation process. The YAML specification format represents the model introduced in Sect. 5.4 in a human- and machine- readable way. To rely on the proposed declarative performance test automation solution, users need to have a basic understanding of the performance testing domain and performance test specification. Users are expected to define a test using the proposed YAML specification format and specify:

1) The test goal and its configuration, as well as the performance space to be explored if required by the goal;

2) The load function;

3) The performance metrics of interest;

4) The criteria to determine if the test execution has to be terminated before its completion;

5) The expected outcome of the test according to defined quality criteria;

6) The workloads used for the test and how to mix its operations;

7) The system under test type, and its configuration;

8) The data collection services needed to collect performance data useful to compute the metrics of interest.

Test specifications relying on the proposed DSL, are expected to be stored alongside the code in the VCS repository where the SUT services are developed. In the following, we report the YAML specification for all the elements the user is expected to define, split for readability purposes. In Listing A.1, part of the App. A we report the entire specification as a single document.

```
version: { String : <Test DSL version, e.g. '1'> }
name: { String : <name of the performance test> }
# OPTIONAL
description: { String : <description of the performance test> }
# OPTIONAL
labels: { [String] : <a list of labels> }
configuration:
  goal:
    ...
  load_function:
    # OPTIONAL IF SPECIFIED IN EXPLORATION
    users: { Number : <total number of users to be simulated> }
    ramp_up: { String : <amount of time><unit> }
    steady_state: { String : <amount of time><unit> }
    ramp_down: { String : <amount of time><unit> }
  # OPTIONAL
  termination_criteria:
    ...
  # OPTIONAL
  quality_gates:
    ...
sut:
  ...
workloads:
  ...
# OPTIONAL
data_collection:
  ...
```

Listing 5.1. DSL: The Test YAML Format Specification

In Listing 5.1 we report the YAML specification of the test model presented in Fig. 5.16. We omit some sections of the model, identified by three dots (...). We also report, using comments, when some parts of the specification are optional (e.g., line #3) according to the referenced model. Each element of the referenced model is represented with a YAML object. Compositions are mapped to object nesting, e.g., the *configuration* (line #7) (*BenchFlowTestConfiguration* in the referenced model) is composed of a single *Goal* (line #8). The nesting of objects helps with understanding the structure of the model, as well as helps in writing the specification with text editors supporting YAML formatting, because usually the editors support collapsing the documentation at each object level, making it easier to write it down.

```
1  configuration:
2   goal:
3     type: { String : < "load" | "smoke" | "sanity" | "configuration" |
      ↪  "scalability" | "spike" | "exhaustive_exploration" | "stability_boundary" |
      ↪  "capacity_constraints" | "regression_complete" | "regression_intersection"
      ↪  | "acceptance" > }
4     # OPTIONAL
5     stored_knowledge: { Boolean : <"true" | "false"> } # default "false"
6     observe:
7       ...
8     # OPTIONAL
9     exploration:
10      ...
```

Listing 5.2. DSL: The Goal YAML Format Specification

In Listing 5.2 we report the YAML specification of the model in Fig. 5.17. The user states the goal of the test, by selecting the goal *type* (line #3). The actual values for the type of test can vary. Usually, also the variant of the name with the addition of the suffix "_test" is supported. If the user wants to reuse *stored knowledge* from previous tests, she can specify that at line #5. If she does not specify the field, we assume by default no stored knowledge shall be reused.

```
1  goal:
2    # OPTIONAL
```

```
3   exploration:
4     exploration_space:
5       services:
6         { String : <name of service> }:
7             # OPTIONAL
8             resources:
9               cpu:
10                # EITHER specific values
11                values: { [String] : <a list of values + unit> }
12                # OR range
13                range: { [String, String] : <a list of values + unit (inclusive)>
                  ↪  }
14                # IF range we can specify step
15                step: { String : <step between the values in the range as a
                  ↪  mathematical expression: +,-,*,/,^><values + unit> }
16              memory:
17                # EITHER specific values
18                values: { [String] : <a list of bytes value + unit> }
19                # OR range
20                range: { [String, String] : <a list of bytes value + unit
                  ↪  (inclusive)> }
21                # IF range we can specify step
22                step: { String : <step between the values in the range as a
                  ↪  mathematical expression: +,-,*,/,^><bytes value + unit> }
23            # OPTIONAL
24            configuration:
25              { String : <name of environment variable> }: { [String] : <a list of
                ↪  possible values> }
26              ...
27        ...
28      load_function:
29        ...
30      # OPTIONAL IF goal.type = stability_boundary
31      stability_criteria:
32        services:
33          { String : <name of service> }:
34            # OPTIONAL
35            avg_cpu: { String : <a mathematical expression:
                  ↪  >,<,>=,<=,=><number><"%"> }
36            # OPTIONAL
37            avg_memory: { String : <a mathematical expression:
                  ↪  >,<,>=,<=,=><number><"%"> }
38          ...
39        workloads:
40          { String : <name of workload> }:
41          ...
```

```
42      exploration_strategy:
43        selection: { String : <"one_at_a_time" | "random_breakdown" |
           ↪  "stability_boundary_first"> }
44        # OPTIONAL
45        validation: { String : <random_validation_set> }
46        # OPTIONAL
47        regression: { String : <mars> }
```

Listing 5.3. DSL: The Exploration Space and Exploration Strategy YAML Format Specification

In Listing 5.3 we report the YAML specification of the model in Fig. 5.18. As reported in line #6 the user can specify a list of named services. For services, she can configure the performance space to be explored either by defining specific values (line #11) or by defining a range of value (e.g., line #13) and a step function to navigate the defined range (e.g., line #15). When specific goals are selected, for example in the case of the *stability_boundary*, specific criteria are specified using a customized part of the YAML specification (e.g., lines #31-#41). Alongside the exploration space, at lines #42-#47 we show how the user can configure the exploration strategy to be used to explore the performance space.

```
1   goal:
2     observe:
3      # OPTIONAL
4      workloads:
5        # OPTIONAL
6        { String : <name of workload> }: { [String] : <a list of workload metrics> }
7        # OPTIONAL
8        { String : <name of workload.operation> }: { [String] : <a list of workload
          ↪  metrics> }
9        ...
10     # OPTIONAL
11     services:
12       { String : <name of service> }: { [String] : <a list of service metrics> }
13       ...
```

Listing 5.4. DSL: The Observe YAML Format Specification

In Listing 5.4 we report the YAML specification of the model in Fig. 5.20.

```
configuration:
 # OPTIONAL
 termination_criteria:
   # OPTIONAL
   test:
     max_time: { String : <amount of time><unit> }
     # OPTIONAL
     max_number_of_experiments: { Number : <maximum number of experiments> }
     # OPTIONAL
     max_failed_experiments: { String : <number><"%"> }
   # OPTIONAL
   experiment:
     max_number_of_trials: { Number : <maximum number of trials> }
     # OPTIONAL
     max_failed_trials: { String : <number><"%"> }
     # OPTIONAL
     workloads:
       # OPTIONAL
       { String : <name of workload> }:
         confidence_interval_metric: { String : <a workload metrics> }
         confidence_interval_value: { Number : <value of the workload metric> }
         confidence_interval_precision: { String : <number><"%"> }
       ...
     # OPTIONAL
     services:
       { String : <name of service> }:
         confidence_interval_metric: { String : <a service metrics> }
         confidence_interval_value: { Number : <value of the service metric> }
         confidence_interval_precision: { String : <number><"%"> }
       ...
```

Listing 5.5. DSL: The Terminationa Criteria YAML Format Specification

In Listing 5.1 we report the YAML specification of the model in Fig. 5.16.

```
configuration:
 # OPTIONAL
 quality_gates:
   # OPTIONAL IF exploration_strategy.regression = mars
   mean_absolute_error: { String : <number><"%"> }
   # OPTIONAL IF goal.type = regression_complete OR goal.type =
    ↪ regression_intersection
   regression:
```

```
8       # EITHER service name
9       service: { String : <name of service> }
10      # OR workload name
11      workload: { String : <name of workload> }
12      gate_metric: { String : <name of the workload or service metric> }
13      # OPTIONAL
14      regression_delta_absolute: { String : <amount><unit> }
15      # OPTIONAL
16      regression_delta_percent: { String : <number><"%"> }
17    # OPTIONAL
18    workloads:
19      { String : <name of workload> }:
20          # OPTIONAL
21        - max_mix_deviation: { String : <number><"%"> }
22          # OPTIONAL
23          max_think_time_deviation: { String : <number><"%"> }
24          # OPTIONAL
25          gate_metric: { String : <name of the workload metric> }
26          # OPTIONAL
27          condition: { String : <a mathematical expression: >,<,>=,<=,=,+%,-%> }
28          # OPTIONAL
29          gate_threshold_target: { String : <a threshold value> OR Number : <value
             ↪  of the workload metric> }
30          # OPTIONAL
31          gate_threshold_minimum: { String : <a threshold value> OR Number : <value
             ↪  of the workload metric> }
32        ...
33      ...
34    # OPTIONAL
35    services:
36      { String : <name of service> }:
37        - gate_metric: { String : <name of the service metric> }
38          condition: { String : <a mathematical expression: >,<,>=,<=,=,+%,-%> }
39          gate_threshold_target: { String : <a threshold value> OR Number : <value
             ↪  of the service metric> }
40          # OPTIONAL
41          gate_threshold_minimum: { String : <a threshold value> OR Number : <value
             ↪  of the service metric> }
42        ...
43      ...
```

Listing 5.6. DSL: The Quality Gates YAML Format Specification

In Listing 5.6 we report the YAML specification of the model in Fig. 5.22.

```
workloads:
 { String : <name of the workload> }
   # OPTIONAL IF more than one workload
   popularity: { String : <number><"%"> }
   { String : <name of the workload item> }
     driver_type: { String : <"start_bpmn" | "http"> }
     # OPTIONAL IF goal.type = regression_complete OR goal.type =
     ↪ regression_intersection
     sut_version: { String : <sut version> }
     # OPTIONAL
     target_service: { String : <name of service> } # default to
     ↪ default_target_service
     # OPTIONAL
     inter_operation_timings:  { String : <"negative_exponential" | "uniform" |
     ↪ "fixed_time"> }
     # OPTIONAL IF more than one workload item
     popularity: { String : <number><"%"> }
     operations:
       ...
     # OPTIONAL IF driver_type = http (Web Services)
     data_sources:
       ...
     # OPTIONAL
     mix:
       ...
   ...
 ...
```

Listing 5.7. DSL: The Workload YAML Format Specification

In Listing 5.7 we report the YAML specification of the model in Fig. 5.23.

```
workloads:
 { String : <name of the workload> }
   # OPTIONAL IF more than one workload
   popularity: { String : <number><"%"> }
   { String : <name of the workload item> }
     # IF driver_type = start_bpmn (WfMS)
     operations:
       - { String : <name of the .bpmn file with the process> }
       ...
     # IF driver_type = http (Web Services)
     operations:
```

```
12        { String : <Name of operation> }
13          protocol: { String : <"http" | "https"> }
14          endpoint: { String : <path to call optionally referencing to data in
             ↪  body, parameter or extracted from response to other operations> }
15          # REFERENCE:
             ↪  https://www.w3.org/Protocols/rfc2616/rfc2616-sec5.html#sec5.1.1
16          method: { String : <"OPTIONS" | "GET" | "HEAD" | "POST" | "PUT" |
             ↪  "DELETE" | "TRACE" | "CONNECT"> }
17          # OPTIONAL
18          # REFERENCE: https://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html
19          headers:
20            Accept: { String : <Media Type> }
21            ...
22          # OPTIONAL
23          query_parameter:
24            - { String : <name of the query parameter> }:
25              items:
26                - { String : <content value> }
27                ...
28              # OPTIONAL
29              retrieval: { String : <"sequential" | "random"> }
30            ...
31          # OPTIONAL
32          url_parameter:
33            - { String : <name of the url parameter> }:
34              items:
35                - { String : <content value> }
36                ...
37              # OPTIONAL
38              retrieval: { String : <"sequential" | "random"> }
39            ...
40          # OPTIONAL
41          extract_regex:
42            - { String : <name assigned to the extracted value> }:
43              # REFERENCE: https://developer.mozilla
                 ↪  .org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/match
44              pattern: { String : <a regex pattern to extract values> }
45              # OPTIONAL
46              default: { String : <the default value assigned when no values are
                 ↪  extracted> }
47              # OPTIONAL
48              match_number: { Number : <the matched element index to select> }
49            ...
50          # OPTIONAL
51          extract_json:
52            - { String : <name assigned to the extracted value> }:
```

```
            # REFERENCE: https://goessner.net/articles/JsonPath/
            pattern: { String : <a JSON selector pattern to extract values> }
            # OPTIONAL
            default: { String : <the default value assigned when no values are
            ↪  extracted> }
            # OPTIONAL
            match_number: { Number : <the matched element index to select> }
          ...
        # OPTIONAL
        body:
          content:
            - { String : <content value> }
            ...
          # OPTIONAL
          retrieval: { String : <"sequential" | "random"> }
        # OPTIONAL
        body_file: { String : <name of a data_sources> }
        # OPTIONAL
        body_form:
          - { String : <name of the form field> }:
            items:
              - { String : <content value> }
              ...
            # OPTIONAL
            retrieval: { String : <"sequential" | "random"> }
          ...
    # OPTIONAL IF driver_type = http (Web Services)
    data_sources:
      - path: { String : <path to a file containing data> }
        delimiter: { String : <delimiter used in the file containing data> }
        # OPTIONAL
        name: { String : <name assigned to the data source> }
        # OPTIONAL
        retrieval: { String : <"sequential" | "random"> }
      ...
    # OPTIONAL
    mix:
      ...
  ...
...
```

Listing 5.8. DSL: The Operations YAML Format Specification

In Listing 5.8 we report the YAML specification of the model in Fig. 5.24.

Although the overall DSL follows a declarative approach, for operations we opt for a more imperative definition. We do so because operations specifications require many details and are often already defined in other formats and languages using imperative approaches, for example, Swagger[9], adopting imperative specifications as well. In Chap. 8 we present a use-case where we abstracted away from the effort of specifying the operations section for the user, by automatically generating it.

```
workloads:
 { String : <name of the workload> }
   # OPTIONAL IF more than one workload
   popularity: { String : <number><"%"> }
   { String : <name of the workload item> }
     # OPTIONAL
     mix:
       # IF FixedSequenceMix
       fixed_sequence: { [String] : <name of operations in the wanted order> }
       # IF FlatMix. List of percentages (SUM = 100%) and optionally think time
       ↪ (tt), index refers to index of operation
       flat: { [String] : <number><"%"> <tt({ Number : <mean> } { Number :
       ↪ <deviation> })> }
       # IF FlatSequenceMix. List of percentages (SUM = 100%) and optionally think
       ↪ time (tt), index refers to index of sequence specified below
       flat: { [String] : <number><"%"> <tt({ Number : <mean> } { Number :
       ↪ <deviation> })> }
         sequences:
           - { [String] : <name of operations in the wanted order> }
           ...
       # IF MatrixMix (needs to be a square matrix). List of percentages (SUM =
       ↪ 100%) and optionally think time (tt), index refers to index of
       ↪ operation
       matrix:
         - { [String] : <number><"%"> <tt({ Number : <mean> } { Number :
         ↪ <deviation> })> }
         ...
   ...
...
```

Listing 5.9. DSL: The Mix YAML Format Specification

In Listing 5.9 we report the YAML specification of the model in Fig. 5.25.

[9] https://swagger.io/, last visited on February 7, 2021

For complex workloads, the specification of the mix can become complex and bloated. In the use-case in which we present a way to automatically generate the workload and operations presented in Chap. 8, we also generate the *mix*.

```
sut:
 name: { String : <name of the sut> }
 versions:
   # EITHER specific values
   values: { [String] : <name of versions in the wanted order> }
   # OR range
   range: { [String, String] : <a list of versions (inclusive)> }
 # OPTIONAL
 type: { String : <"wfms" | "http"> }
 sut_configuration:
   default_target_service:
     name: { String : <name of target service> }
     endpoint: { String : <base endpoint for each operation> }
     # OPTIONAL
     sut_ready_log_check: { String : <a regular expression to check the
     ↪  availability of the sut> }
   # OPTIONAL
   deployment:
     { String : <name of service> }: { String : <name of server alias> }
     ...
 # OPTIONAL
 services_configuration:
   { String: <name of service> }:
     # OPTIONAL
     resources:
       # OPTIONAL
       cpu: { [String] : <a list of values + unit> }
       # OPTIONAL
       memory: { [String] : <a list of bytes value + unit> }
     # OPTIONAL
     configuration:
       { String: <name of environment variable> }: { String: <value> }
       ...
   ...
```

Listing 5.10. DSL: The SUT YAML Format Specification

In Listing 5.10 we report the YAML specification of the model in Fig. 5.26.

```
1  # OPTIONAL
2  data_collection:
3   only_declared: { Boolean : <"true" | "false"> } # default "false"
4   # OPTIONAL
5   services:
6     # IF collector does NOT require CONFIGURATION
7     { String : <name of service> }: { String | [String] : <names of BenchFlow
       ↪  collectors> }
8     # IF some collectors requires CONFIGURATION
9     { String : <name of service> }:
10      { String : <name of BenchFlow collector> }:
11        configuration:
12          { String : <name of environment variable> }: { [String] : <a list of
           ↪  possible values> }
13          ...
14      ...
15    ...
16  # OPTIONAL
17  workloads:
18    { String : <name of the workload> }:
19      # OPTIONAL IF trial execution framework = JMeter
20      jmeter:
21        interval: { String : <time interval for data collection><"s" (seconds)> } #
           ↪  default 1s
22      # OPTIONAL IF trial execution framework = Faban
23      faban:
24        interval: { String : <time interval for data collection><"s" (seconds)> } #
           ↪  default 1s
25    ...
```

Listing 5.11. DSL: The Data Collection YAML Format Specification

In Listing 5.11 we report the YAML specification of the model in Fig. 5.28. The user only needs to specify data collection services requiring configurations (e.g., the ones defined by the specification at lines #8-#15). The other collector services can be inferred by the framework part of the proposed approach, and automatically added to the specification according to the *Observe*d metrics. The framework part of the proposed approach could also decide to activate additional collector services, to collect more performance data than the ones needed for computing the observed metrics. This behavior can be disabled, by setting the *only_declared* field (line #3) to *true*.

```
version: { String : <Experiment DSL version, e.g. "1"> }
name: { String : <name of the performance experiment derived from the name of the test> }
# OPTIONAL
description: { String : <description of the performance experiment derived from the name of the test> }
# OPTIONAL
labels: { [String] : <a list of labels> }
configuration:
  # SET dependent on the load_function or exploration specification at test level
  load_function:
    users: { Number : <total number of users to be simulated> }
    ramp_up: { String : <amount of time><unit> }
    steady_state: { String : <amount of time><unit> }
    ramp_down: { String : <amount of time><unit> }
  termination_criteria:
    max_time: { String : <amount of time><unit> }
    # OPTIONAL
    experiment:
      # SAME as per the Test
      ...
sut:
  name: { String : <name of the sut> }
  version: { String : <name of version> }
  # OPTIONAL
  type: { String : <"wfms" | "http"> }
  sut_configuration:
    # SAME as per the Test
    ...
  # OPTIONAL. SET dependent on the load_function or exploration specification at test level
  services_configuration:
    { String: <name of service> }:
      # OPTIONAL
      resources:
        # OPTIONAL
        cpu: { [String] : <a list of values + unit> }
        # OPTIONAL
        memory: { [String] : <a list of bytes value + unit> }
      # OPTIONAL
      configuration:
        { String: <name of environment variable> }: { String: <value> }
        ...
    ...
workloads:
```

```
  # SAME as per the Test
  ...
# DEPENDENT on the observe and the data_collection sections in the test
 ↪  specification
data_collection:
  only_declared: { Boolean : <"true" | "false"> } # default "false"
  # OPTIONAL.
  services:
    # SAME as per the Test
    ...
  # OPTIONAL
  workloads:
    # SAME as per the Test
    ...
```

Listing 5.12. DSL: The Experiment YAML Format Specification

In Listing 5.12 we report the YAML specification of the experiment model in Fig. 5.29. We omit all the entities that are the same as per the test YAML specification. In Listing A.2, part of the App. A we report the entire specification as a single document. The user is not expected to specify the experiment model, it is generated by the framework part of the proposed approach.

5.6.2 DSL for CSDL YAML Complete Specification

```
version: { String : <Test Suite CDSL version, e.g. '1'> }
name: { String : <name of the performance test suite> }
# OPTIONAL
description: { String : <description of the performance test suite> }
suite:
  triggers:
    # OPTIONAL
    scheduled: { Boolean : <"true" | "false"> } # default "false"
    # OPTIONAL
    on:
      # OPTIONAL
      push:
        # OPTIONAL
        branches:
          - { String : <expression matching the name of branch> }
          ...
```

```
      # OPTIONAL
      pull_request:
        # OPTIONAL
        contexts: { String : <"head" | "merge" | "all"> }
        # OPTIONAL
        source_branches:
          - { String : <expression matching the name of branch> }
          ...
        # OPTIONAL
        target_branches:
          - { String : <expression matching the name of branch> }
          ...
      # OPTIONAL
      releases:
        types: { [String] : <types of releases> }
      # OPTIONAL
      deployments: { [String] : <names of deployments> }
  # OPTIONAL IF DEPLOYMENTS
  environments:
    - { String : <name of environments> }:
        # OPTIONAL
        skip_deploy: { Boolean : <"true" | "false"> } # default "false"
    ...
  tests: -
    # EITHER a list of file paths containing tests
    - { String : <path of file containing test> }
    ...
    # OR
    include_labels: { [String] : <expression matching the name of labels to select
    ↪  tests> }

quality_gates:
  criterion: { String : <"all_success" | "at_least_one_success"> }
  # OPTIONAL
  exclude:
    - { String : <path of file containing test> }
    ...
```

Listing 5.13. DSL for CSDL: The Tests Suite YAML Format Specification

In Listing 5.13 we report the YAML specification of the CSDL model in Fig. 5.32. The user is expected to define multiple specifications of test suites according to the presented YAML format, for the different test suites and moments of the CSDL in which she plan to execute them.

5.6.3 YAML Examples

In this section, we present different actual YAML specification examples for the different elements of the DSL and complete specifications of tests. The different specification examples all belong to a complex *STABILITY_BOUNDARY* performance test we completely report in Listing B.1 in App. B. We opted for the *STABILITY_BOUNDARY* test as an example, because it is a complex test, requiring most of the elements of the YAML specification presented in Sect. 5.6.1.

DSL YAML Examples

```yaml
version: "3"
name: "Predictive Stability Boundary Test"
description: "Example of Predictive Stability Boundary Test re-using Store
↪ Knowledge and Observing Client- and Server-side Metrics"
labels: "stability_boundary", "predictive", "stored_knowledge"
configuration:
  goal:
    type: "stability_boundary"
    stored_knowledge: "true"
    observe:
      ...
    exploration:
      exploration_space:
        ...
        stability_criteria:
          ...
      exploration_strategy:
        ...
  load_function:
    ...
  termination_criteria:
    test:
      ...
    experiment:
      ...
  quality_gates:
    ...
sut:
  ...
workloads:
  ...
```

```
data_collection:
  ...
```

Listing 5.14. DSL: The Test YAML Skeleton Example

In Listing 5.14 we report an example of a complete skeleton of a performance test. The skeleton in the example represents a *STABILITY_BOUNDARY* performance test as evident from the *type* specification at line #7. The specification states the reuse of stored knowledge (line #8), important in this case because stability boundary tests are usually quite complex and time-consuming. *Labels* are also provided (line #4).

```
configuration:
 goal:
   type: "stability_boundary"
   stored_knowledge: "true"
   observe:
     workloads:
       workload_a: avg_response_time, avg_latency
       workload_b: avg_response_time, avg_latency
       workload_b.operation_a: avg_response_time
     services:
       service_a: avg_cpu, avg_memory
       service_b: avg_cpu, avg_memory
       dbms_a: avg_cpu, avg_memory, avg_io
   exploration:
     exploration_space:
       services:
         service_a:
           resources:
             cpu:
               range: [100m, 1000m]
               step: "*4"
             memory:
               range: [256Mi, 1024Mi]
               step: "+768Mi"
           configuration:
             NUM_SERVICE_THREAD: [12, 24]
         dbms_a:
           resources:
             cpu:
               range: [100m, 1000m]
```

```
31              step: "*10"
32            memory:
33              range: [256Mi, 1024Mi]
34              step: "+768Mi"
35          configuration:
36            QUERY_CACHE_SIZE: 48Mi
37      stability_criteria:
38        services:
39          service_a:
40            avg_cpu: "<=60%"
41            avg_memory: "<=80%"
42          dbms_a:
43            avg_memory: "<=70%"
44        workloads:
45          workload_b:
46            max_mix_deviation: 5%
47    exploration_strategy:
48      selection: "stability_boundary_first"
49      validation: "random_validation_set"
50      regression: "mars"
```

Listing 5.15. DSL: The Goal YAML Example

In Listing 5.15 we report an example of a *Goal* section configuration related to the *STABILITY_BOUNDARY* test of Listing 5.14. The stability boundary test in the example states:

a) the *stability_boundary* test goal, being the specified test a stability boundary test (line #3);

b) the metrics to *Observe*, on workloads, operations, and services (lines #5-#13). The user is interested in observing *avg_response_time* on "workload_a" and "workload_b", and *avg_latency* on "operation_a" of "workload_b". She is also interested in *avg_cpu* and *avg_memory* for "service_a", "service_b", and "dbms_a", as well as *avg_io* for "dbms_a";

c) the *Exploration Space* at lines #15-#36, involving "service_a" and "dbms_a", on which the user explores *CPU*, *RAM* and values of configuration variables (lines #26 and #36). The exploration CPU (*cpu*), and *RAM* (*memory*) is defined using a *range* and a *step* function.

d) the *stability_criteria* at lines #37-#46 needed according to the test goal type. The stability criteria are defined on both services and workloads,

and involves “service_a”, “dbms_a”, and “workload_b”. The user is interested in ensuring the workload issued to the system does not deviate more than 5% from the specification (line #46) and CPU and/or RAM for the two involved services must remain on average under the specified thresholds;

e) the *exploration_strategy* at line #47-#50, where the user selects the *stability_boundary_first* selection criteria, coherently with the type of test, and specifies she wants to rely on the *mars* predictive exploration, selecting the validation set for validating the precision of the model at random (*random_validation_set*).

Given the way the exploration space is generated out of the specification, as presented in Sect. 5.4, for a stability boundary test the user specifies the values for the exploration space from the expected worst performing one, to the expected best-performing ones. As for example, for CPU the user defines the range as starting from providing the less CPU to the services (i.e., *100m*) to the most CPU (i.e., *1000m*).

```
1 configuration:
2  load_function:
3    users: 1000
4    ramp_up: 5m
5    steady_state: 20m
6    ramp_down: 5m
```

Listing 5.16. DSL: The Load Function YAML Example

In Listing 5.16, we report an example of a `load_function`. The load function is defined to simulate 1000 users, according to a given load function shape based on time. We use an underlying `java.time.Duration` type to store the time, and then use the `java.time.temporal.TemporalUnit` for the time unit. In this way, we can store the original time unit the user specifies as well as dynamically convert to other time units as needed, which is important to be able to provide flexibility to the users, as well as to convert to and from other time specifications used in other parts of the framework part of the proposed approach and integrated trial execution frameworks.

```yaml
configuration:
 termination_criteria:
   test:
     max_time: 120h
     max_failed_experiments: 10%
   experiment:
     max_failed_trials: 10%
     workloads:
       workload_a:
         confidence_interval_metric: avg_response_time
         confidence_interval_value: 50ms
         confidence_interval_precision: 95%
     services:
       service_a:
         confidence_interval_metric: avg_cpu
         confidence_interval_value: 60%
         confidence_interval_precision: 95%
```

Listing 5.17. DSL: The Termination Criteria YAML Example

In Listing 5.17 we report an example of termination criteria. The example is related to the `STABILITY_BOUNDARY` test. The termination criteria are defined considering the expected long execution time of the test, according to the defined exploration space and the load function. For the test, the user defines a termination criterion based on the maximum execution time the test is allowed to execute (line #4), and a criterion on the maximum number of failing experiments (line #5). The experiment termination criteria control the maximum percentage of allowed failing trials (line #7), and define thresholds for the `avg_response_time` and `avg_cpu` for a workload and a service respectively. The thresholds are consistent with the `stability_criteria` defined in Listing 5.15. The framework part of the proposed approach checks for consistency of the termination criteria, with the number and time needed for executing the experiments part of the test and warns the user in case of inconsistencies, e.g., on the total time a test is allowed to execute.

```yaml
configuration:
 quality_gates:
   mean_absolute_error: 10%
   workloads:
     workload_a:
```

```
6         - max_mix_deviation: 5%
7           max_think_time_deviation: 2%
8           gate_metric: avg_response_time
9           condition: "<="
10          gate_threshold_target: "100ms"
11          gate_threshold_minimum: "200ms"
12    services:
13      service_a:
14        - gate_metric: avg_cpu
15          condition: "<="
16          gate_threshold_target: 50%
17          gate_threshold_minimum: 60%
```

Listing 5.18. DSL: The Quality Gates YAML Example

In Listing 5.18 we report an example of quality gates related to the `STABILITY_BOUNDARY` test. Given the `exploration_strategy` in Listing 5.15 selected for the test is a predictive strategy, the user defines the `mean_absolute_error` (line #3) accepted to consider the built model as satisfactory in representing the performance on the SUT in the defined performance space. Other quality gates are defined both on a workload (lines #5-#11) and a service (lines #13-#17) metric. The quality gates define when to consider the test successful, in the points of the exploration space where the `stability_criteria` defined in Listing 5.15 are satisfied.

```
1  sut:
2   name: "my_app"
3   versions:
4     values: "v1.5"
5   type: "http"
6   sut_configuration:
7     default_target_service:
8       name: "service_a"
9       endpoint: "/"
10      sut_ready_log_check: "/(.*)System started(.*)/g"
11    deployment:
12      service_a: "my_server"
13  services_configuration:
14    service_b:
15      resources:
16        cpu: 200m
17        memory: 256Mi
```

```
18      configuration:
19        THREADPOOL_SIZE: 64
```

Listing 5.19. DSL: The SUT YAML Example

In Listing 5.19 we report the definition of the SUT configuration for the test. Among the different data specified, the user indicates the SUT is of `type: ``http''` (line #5), the default target service is the `service_a` and for the SUT to be considered ready for the test, the `service_a` log has to contain a string matched by the following regular expression ``/(.*)System started(.*)/g'' (lines #7-#10). The user also states the `service_a` has to be deployed on a sever identified by "my_server" (line #10) and specifies some configuration options for the `service_b` at lines #14-#19.

```
1  workloads:
2   workload_a:
3     popularity: 70%
4     item_a:
5       driver_type: "http"
6       inter_operation_timings: "negative_exponential"
7       popularity: 80%
8       operations:
9         operation_a:
10          protocol: "https"
11          endpoint: "/"
12          method: "GET"
13          extract_regex:
14            - title:
15              pattern: "<title>(.*?)</title>"
16              default: ""
17              match_number: 1
18        operation_b:
19          protocol: "https"
20          endpoint: "/${title}"
21          method: "POST"
22          body_file: "datasource_a"
23        data_sources:
24          - path: "/path_to_datasource_a"
25            delimiter: ","
26            name: datasource_a
27            retrieval: "random"
28        mix:
```

```
29          flat: "75.0% tt(1000.0 500.0), 25.0% tt(2000.0 400.0)"
30      item_b:
31        driver_type: "http"
32        inter_operation_timings: "negative_exponential"
33        popularity: 20%
34        operations:
35          operation_a:
36            protocol: "https"
37            endpoint: "/"
38            method: "GET"
39    workload_b:
40      popularity: 30%
41      item_a:
42        driver_type: "http"
43        target_service: "service_b"
44        operations:
45          operation_a:
46            protocol: "https"
47            endpoint: "/"
48            method: "GET"
```

Listing 5.20. DSL: The Workloads, Workload Items, and Operations YAML Example

In Listing 5.20 we present the workload the user specifies for the test, along with workload items and operations. The user specifies two workloads, `workload_a` at lines #2-#38, and `workload_b` at lines #39-#48. The user defines she wants to execute `workload_a` 70% of the times (line #3), while `workload_b` the remaining 30% of the times (line #40). `workload_a` has two workload items, `item_a` at lines #4-#29, and `item_b` at lines #30-#38. The two items have a 80% (line #7) and 20% (line #33) probability of being executed respectively, and both follow a `negative_exponential` distribution (line #6 and line #32). The `popularity` defines the "popularity" of the workload item, i.e., how many of the virtual users, simulating the workload where the workload item belongs to, perform the operations of the given workload item during the test execution. For example, if the number of simulated users is 1000, the workload of interest simulates 70% of the users and there are two workload items specified with 80% and 20% popularity respectively, 560 virtual users will simulate the operations of the first workload item, and the remaining 240 will simulate the operations of the second one. `item_a` defines two operations (lines #8-#29). `operation_a` extract data from the response using a regular expression (lines

#13-#17) while `operation_b` relies on a `data_source` to retrieve data to be used as part of the interaction with the SUT (line #22, referencing the data source at lines #23-#27). The two operations are executed according to a `flat mix` defining `operation_a` is chosen to be executed as next operation 75% of the times, while `operation_b` 25% of the times. The user also reports the expected think time of the simulated users before deciding which next operation to execute. `workload_b` has only one item targeting `service_b` (line #43).
Other formats for specifying a mix for the example in Listing 5.20, according to the model in Fig. 5.25 are:

a) a fixed sequence mix (Listing 5.21), that in the case of the example specifies a fixed sequence of calls where `operation_a` is always called before `operation_b`;

b) a flat sequence mix (Listing 5.22), that in the case of the example specifies two sequences (lines #2-#4), one where `operation_a` is always called before `operation_s` and the second one where the opposite happens. The first sequence is expected to be executed 95% of the times, while the second one 5% on the times;

c) a matrix mix (Listing 5.23), that in the case of the example specifies: a probability of 75% of calling the `operation_a` again after executing `operation_a`, while a 25% of calling the `operation_b` after calling `operation_a` (line #2), and a probability of 25% of calling the `operation_a` after executing `operation_b`, while a 75% of calling the `operation_b` again after calling `operation_b` (line #3);

```
1 fixed_sequence: operation_a, operation_b
```

Listing 5.21. DSL: Fixed Sequence Mix YAML Example

```
1 flat: "95.0% tt(1500.0 400.0), 5.0% tt(1200.0 600.0)"
2  sequences:
3    - operation_a, operation_b
4    - operation_b, operation_a
```

Listing 5.22. DSL: Flat Sequence Mix YAML Example

```yaml
matrix:
 - [ 75.0% tt(1000.0 500.0), 25.0% tt(2000.0 400.0) ]
 - [ 25.0%, 75.0% tt(2000.0 400.0) ]
```

Listing 5.23. DSL: Matrix Mix YAML Example

```yaml
data_collection:
 # AUTOMATICALLY attached based on the observe section IF NOT specified
 services:
   service_a: "stats"
   service_b: "stats"
   dbms_a: "stats"
```

Listing 5.24. DSL: The Data Collection YAML Example

In Listing 5.24 we report an example of specification for data collector services. In the case of the example, no collection service requires configuration, thus the specification reported in Listing 5.24 could also be generated by the framework part of the proposed approach according to the *Observe* section in Listing 5.15.

```yaml
version: "3"
name: "Predictive Stability Boundary Test - Experiment 1"
description: "Example of Predictive Stability Boundary Test re-using Store
↪  Knowledge and Observing Client- and Server-side Metrics"
labels: "stability_boundary", "predictive", "stored_knowledge"
configuration:
  load_function:
    users: 1000
    ramp_up: 5m
    steady_state: 20m
    ramp_down: 5m
  termination_criteria:
    max_time: 150m
    experiment:
      max_number_of_trials: 10
      max_failed_trials: 10%
      workloads:
        workload_a:
          confidence_interval_metric: avg_response_time
          confidence_interval_value: 50ms
```

```yaml
          confidence_interval_precision: 95%
      services:
        service_a:
          confidence_interval_metric: avg_cpu
          confidence_interval_value: 60%
          confidence_interval_precision: 95%
sut:
  name: "my_app"
  version: "v1.5"
  type: "http"
  sut_configuration:
    default_target_service:
      name: "service_a"
      endpoint: "/"
      sut_ready_log_check: "/(.*)System started(.*)/g"
    deployment:
      service_a: "my_server"
  services_configuration:
    service_a:
      resources:
        cpu: 100m
        memory: 256Mi
      configuration:
        NUM_SERVICE_THREAD: 12
    service_b:
      resources:
        cpu: 200m
        memory: 256Mi
      configuration:
        THREADPOOL_SIZE: 64
    dbms_a:
      resources:
        cpu: 100m
        memory: 256Mi
      configuration:
        QUERY_CACHE_SIZE: 48Mi
workloads:
  workload_a:
    popularity: 70%
    item_a:
      driver_type: "http"
      inter_operation_timings: "negative_exponential"
      popularity: 80%
      operations:
        operation_a:
          protocol: "https"
```

```
          endpoint: "/"
          method: "GET"
          extract_regex:
            - title:
              pattern: "<title>(.*?)</title>"
              default: ""
              match_number: 1
        operation_b:
          protocol: "https"
          endpoint: "/${title}"
          method: "POST"
          body_file: "datasource_a"
        data_sources:
          - path: "/path_to_datasource_a"
            delimiter: ","
            name: datasource_a
            retrieval: "random"
        mix:
          flat: "75.0% tt(1000.0 500.0), 25.0% tt(2000.0 400.0)"
    item_b:
      driver_type: "http"
      inter_operation_timings: "negative_exponential"
      popularity: 20%
      operations:
        operation_a:
          protocol: "https"
          endpoint: "/"
          method: "GET"
  workload_b:
    popularity: 30%
    item_a:
      driver_type: "http"
      target_service: "service_b"
      operations:
        operation_a:
          protocol: "https"
          endpoint: "/"
          method: "GET"
data_collection:
  # AUTOMATICALLY attached based on the observe section IF NOT specified
  services:
    service_a: "stats"
    service_b: "stats"
    dbms_a: "stats"
```

Listing 5.25. DSL: The Experiment YAML Example

In Listing 5.25 we report one of the experiment specifications the framework part of the proposed approach automatically generates according to the specification in Listing 5.14. The reported specification corresponds to the first experiment to be scheduled for execution. This is evident from the configuration of the `sut` at lines #37-#55. `service_a` and `dbms_a` are configured selecting the first point in the defined performance exploration space, the one where the SUT is expected to perform the worst. The configuration for `service_b`, on the other end, is always the same according to the one defined in Listing 5.19. The `termination_criteria` at lines #11-#25 are defined according to the one specified at the test level. The `max_time` termination criterion is computed by the framework part of the proposed approach, according to the total number of experiments in the exploration space (48), and the `max_time` the test is allowed to execute defined at line #4 in Listing 5.17. The obtained termination criterion of 150 minutes is consistent with the load function specified in Listing 5.16, accounting for multiple trials for each experiment.

SUT Deployment Descriptor YAML Examples

In this section we report an example of a SUT deployment descriptor specified in YAML according to the model presented in Fig. 5.30.

```yaml
version: "3.8"
services:
  service_a:
    image: service_a:v1.5
    command: "./start_service.sh"
    ports:
      - "443"
    networks:
      - frontend
      - backend
    environment:
      NUM_SERVICE_THREAD: 12
    deploy:
      replicas: 2
      placement:
        constraints:
          - "server_name==my_server"

```

```
19    service_b:
20      image: service_b:v1.5
21      ports:
22        - "443"
23      networks:
24        - frontend
25        - backend
26      environment:
27        THREADPOOL_SIZE: 48
28      deploy:
29        replicas: 2
30
31    dbms_a:
32      image: mysql:8.1
33      volumes:
34        - db-data:/var/lib/mysql/data
35      networks:
36        - backend
37      environment:
38        QUERY_CACHE_SIZE: 32Mi
39      deploy:
40        replicas: 1
41
42  networks:
43    frontend:
44    backend:
45
46  volumes:
47    db-data:
```

Listing 5.26. SUT Deployment Descriptor YAML Example

In Listing 5.26 we report an exemplificatory SUT deployment descriptor that could be specified along the test example provided in Listing 5.14. The deployment descriptor specifies three services, `service_a` at lines #3-#17, `service_b` at lines #19-#29, and `dbms_a` at lines #31-#40. The services rely on `networks` specified at lines #42-#44 and data volumes specified at lines #46-#47. For the `service_a` the user also specifies `placement` constrains according to the server name (lines #15-#17).

DSL Goal YAML Examples

In this section, we list an example of a specification for each of the supported test goals. We omit details already specified in the example in Sect. 5.6.3, and we focus on relevant details given the stated performance test goal.

```yaml
version: "3"
name: "Load Test"
description: "Example of Load Test"
labels: "load_test"
configuration:
  goal:
    type: "load_test"
    # stored_knowledge: "false"
    observe:
      ...
  load_function:
    users: 1000
    ramp_up: 5m
    steady_state: 20m
    ramp_down: 5m
  termination_criteria:
    ...
  quality_gates:
    ...
sut:
  ...
workloads:
  ...
data_collection:
  # AUTOMATICALLY attached based on the observe section IF NOT specified
  services:
    ...
```

Listing 5.27. DSL: A Load Test for a Web service YAML Example

In Listing 5.27 we report an example of load test goal for RESTful Web services. The load test in the example states:

a) the load test goal, being the specified test a load test (line #7);

b) the load function (lines #11-#15).

Usually, a load test does not reuse `stored_knowledge`.

```yaml
version: "3"
name: "Load Test on WfMS"
description: "Example of Load Test on a WfMS"
labels: "load_test", "wfms"
configuration:
  goal:
    type: "load_test"
    # stored_knowledge: "false"
    observe:
      workloads:
        wfms_a: avg_response_time, number_process_instances,
        ↪  avg_process_instance_execution_time
      services:
        wfms_a: avg_cpu, avg_memory
        dbms_a: avg_cpu, avg_memory, avg_io
  load_function:
    users: 1000
    ramp_up: 5m
    steady_state: 20m
    ramp_down: 5m
  termination_criteria:
    ...
  quality_gates:
    ...
sut:
  name: "my_wfms"
  versions:
    values: "v10.4"
  type: "wfms"
  sut_configuration:
    default_target_service:
      name: "wfms_a"
      endpoint: "/"
      sut_ready_log_check: "/(.*)WfMS started(.*)/g"
    deployment:
      wfms_a: "my_server"
  services_configuration:
    wfms_a:
      resources:
        cpu: 1000m
        memory: 2048Mi
      configuration:
        THREADPOOL_SIZE: 128
workloads:
  workload_a:
```

```
45      item_a:
46        driver_type: "wfms"
47        inter_operation_timings: "negative_exponential"
48        operations:
49          - bpmn_process_a.bpmn20
50          - bpmn_process_b.bpmn20
51          mix:
52            fixed_sequence: bpmn_process_a.bpmn20, bpmn_process_b.bpmn20
53  data_collection:
54    # AUTOMATICALLY attached based on the observe section IF NOT specified
55    services:
56      wfms_a: "stats"
57      dbms_a:
58        mysql:
59          configuration:
60            MYSQL_DB_NAME: process_instances_database_name
61            MYSQL_USER: ${BENCHFLOW_TEST_BOUNDSERVICE_CONFIG_MYSQL_USER}
62            MYSQL_USER_PASSWORD:
              ↪ ${BENCHFLOW_TEST_BOUNDSERVICE_CONFIG_MYSQL_USER_PASSWORD}
63            TABLE_NAMES: ACT_HI_PROCINST, ACT_HI_ACTINST
64            MYSQL_PORT: ${BENCHFLOW_TEST_BOUNDSERVICE_CONFIG_MYSQL_PORT}
65            COMPLETION_QUERY:
              ↪ "SELECT+COUNT(*)+FROM+ACT_HI_PROCINST+WHERE+END_TIME_+IS+NULL"
66            COMPLETION_QUERY_VALUE: 0
67            COMPLETION_QUERY_METHOD: "equal"
```

Listing 5.28. DSL: A Load Test for a WfMSs YAML Example

In Listing 5.28 we report an example of load test goal for a BPMN 2.0 WfMSs. The load test in the example states:

a) the load test goal, being the specified test a load test (line #7);

b) BPMN 2.0 WfMSs specific metrics to observe on the BPMN 2.0 WfMSs service (lines #11 and #13);

c) the load function (lines #15-#19);

d) the `sut` type as `wfmss` (line #28), the `sut_configuration` at lines #29-#35, and the `service_configuration` at lines #36-#42;

e) a `Workload` relying on the WfMSs specific configuration for operations (lines #48-#50);

f) data collection services for `wfms_a` and `dbms_a`. For `dbms_a`, the user specifies a data collection service to retrieve data from a DBMS where the WfMSs stores performance data about the executed process instances. Such collector requires configuration, and in the case of the example the following configuration is defined:

   a) *MYSQL_DB_NAME*, indicating the names of the databases where to find the tables to store;
   b) *MYSQL_USER*, referring to the variable defining a user authorized to access the DBMS, specified in the deployment descriptor of the SUT;
   c) *MYSQL_USER_PASSWORD*, the password of the *MYSQL_USER*;
   d) *MYSQL_PORT*, the port to access the DBMS;
   e) *TABLE_NAMES*, indicating the names of the tables to store;
   f) *COMPLETION_QUERY*, the query executed to verify when the current trial can be considered complete, and performance data can be stored;
   g) *COMPLETION_QUERY_VALUE*, the result value expected for the *COMPLETION_QUERY*;
   h) *COMPLETION_QUERY_METHOD*, the method to compare the result of the *COMPLETION_QUERY* and the *COMPLETION_QUERY_VALUE*.

   More details about available collector services and configuration of the same are provided in Sect. 6.2.5.

```yaml
version: "3"
name: "Smoke Test"
description: "Example of Smoke Test"
labels: "smoke_test"
configuration:
  goal:
    type: "smoke_test"
    # stored_knowledge: "false"
    observe:
      ...
  load_function:
    users: 10
```

```
    ramp_up: 30s
    steady_state: 2m
    ramp_down: 30s
  termination_criteria:
    test:
      max_time: 20m
    ...
  quality_gates:
    ...
sut:
  ...
workloads:
  ...
data_collection:
  # AUTOMATICALLY attached based on the observe section IF NOT specified
  services:
    ...
```

Listing 5.29. DSL: A Smoke Test for a Web service YAML Example

In Listing 5.29 we report an example of a smoke test goal. The smoke test in the example states:

a) the goal type to ”smoke_test” (line #7);

b) the load function (lines #11-#15) and termination criterion (lines #16-#18). Compared to the load test in Listing 5.27 the load function simulates fewer users for a lower amount of time. Smoke tests usually need less time than load tests to be executed.

A smoke test is very similar to a load test in terms of test specification, thus it is possible to reuse load test specifications to define smoke tests by only adjusting the configurations of interest.

```
version: "3"
name: "Sanity Test"
description: "Example of Sanity Test"
labels: "sanity_test"
configuration:
  goal:
    type: "sanity_test"
    # stored_knowledge: "false"
```

```
    observe:
      ...
  load_function:
    users: 10
    ramp_up: 30s
    steady_state: 2m
    ramp_down: 30s
  termination_criteria:
    test:
      max_time: 20m
    ...
  quality_gates:
    ...
sut:
  ...
workloads:
  ...
data_collection:
  # AUTOMATICALLY attached based on the observe section IF NOT specified
  services:
    ...
```

Listing 5.30. DSL: A Sanity Test for a Web service YAML Example

In Listing 5.30 we report an example of a sanity test goal. The test is very similar to a load test one, thus also in this case one can reuse load test specifications. For a sanity test, the user usually would target specific functionalities, by defining a focused workload.

```
version: "3"
name: "Configuration Test"
description: "Example of Configuration Test"
labels: "configuration"
configuration:
  goal:
    type: "configuration"
    stored_knowledge: "true"
    observe:
      ...
    exploration:
      exploration_space:
        services:
          service_a:
```

```
15            resources:
16              cpu:
17                range: [100m, 1000m]
18                step: "*4"
19              memory:
20                range: [256Mi, 1024Mi]
21                step: "+768Mi"
22            configuration:
23              NUM_SERVICE_THREAD: [12, 24]
24          dbms_a:
25            resources:
26              cpu:
27                range: [100m, 1000m]
28                step: "*10"
29              memory:
30                range: [256Mi, 1024Mi]
31                step: "+768Mi"
32            configuration:
33              QUERY_CACHE_SIZE: 48Mi
34      exploration_strategy:
35        selection: "one_at_a_time"
36  load_function:
37    ...
38  termination_criteria:
39    ...
40  quality_gates:
41    ...
42sut:
43  ...
44workloads:
45  ...
46data_collection:
47  ...
```

Listing 5.31. DSL: A Configuration Test for a Web service YAML Example

In Listing 5.31 we report an example of configuration test goal. The type test in the example states:

a) the goal as being a configuration test (line #7);

b) re-utilization of stored knowledge (line #8), important to reduce the execution time;

c) the exploration space, required to specify the space of configuration to explore (lines #11-#33). The exploration space covers multiple services resources and configurations;

d) the exploration strategy (lines #34-#35) as one selecting one experiment at a time in the exploration space following the order of dimensions and value specified in the exploration space section.

The specification for the `termination_criteria` and `quality_gates` would be similar to the ones of a `STABILITY_BOUNDARY` test as reported in Listing 5.17 and Listing 5.18 respectively. When defining `STABILITY_BOUNDARY` tests, it is common to reuse `CONFIGURATION_TEST` specifications.

```
1  version: "3"
2  name: "Scalability Test"
3  description: "Example of Scalability Test"
4  labels: "scalability"
5  configuration:
6    goal:
7      type: "scalability"
8      observe:
9        workloads:
10         workload_a: avg_response_time, avg_latency
11         workload_b: avg_response_time, avg_latency
12       services:
13         service_a: avg_cpu, avg_memory
14         service_b: avg_cpu, avg_memory
15         dbms_a: avg_cpu, avg_memory, avg_io
16     exploration:
17       exploration_space:
18         load_function:
19           users:
20             range: [100, 1000]
21             step: "+100"
22       exploration_strategy:
23         selection: "random_breakdown"
24   load_function:
25     ramp_up: 1m
26     steady_state: 5m
27     ramp_down: 1m
28   termination_criteria:
29     test:
30       max_time: 6h
31       max_failed_experiments: 10%
32     experiment:
```

```yaml
      max_failed_trials: 10%
      workloads:
        workload_a:
          confidence_interval_metric: avg_response_time
          confidence_interval_value: 200ms
          confidence_interval_precision: 95%
  quality_gates:
    workloads:
      workload_a:
        - max_mix_deviation: 5%
          max_think_time_deviation: 2%
          gate_metric: avg_response_time
          condition: "<="
          gate_threshold_target: "100ms"
          gate_threshold_minimum: "200ms"
sut:
  ...
workloads:
  ...
data_collection:
  ...
```

Listing 5.32. DSL: A Scalability Test for a Web service YAML Example

In Listing 5.32 we report an example of scalability test goal. The scalability test in the example states:

a) the goal as being a scalability test (line #7);

b) the metrics to `observe` of interest for the scalability test (lines #8-#15), both on workloads and resource utilization;

c) the exploration space, defined on the load function to generate multiple experiments testing the performance of the SUT under different number of simulated users (lines #16-#21);

d) the exploration strategy to pick at random the experiments in the exploration space (lines #22-#23);

e) the load function (lines #24-#27);

f) the quality gates, ensuring the quality of the SUT during the scalability test respects the criteria defined at lines #40-#47.

g) the termination criteria, defining a `max_time` for the test accounting for the number of experiments to be executed (line #30) and a termination criterion on the `workload_a` to ensure the SUT performs according to the specified response time (lines #35-#38), among other criteria. The termination criterion on `workload_a` is specified according to the defined quality gates so that the test gets terminated in case the quality gate can not be guaranteed during the scalability test.

```
1  version: "3"
2  name: "Spike Test"
3  description: "Example of Spike Test"
4  labels: "spike"
5  configuration:
6    goal:
7      type: "spike"
8      observe:
9        workloads:
10         workload_a: avg_response_time, avg_latency
11         workload_b: avg_response_time, avg_latency
12       services:
13         service_a: avg_cpu, avg_memory
14         service_b: avg_cpu, avg_memory
15         dbms_a: avg_cpu, avg_memory, avg_io
16   load_function:
17     users: 5000
18     ramp_up: 1m
19     steady_state: 10s
20     ramp_down: 1m
21   termination_criteria:
22     test:
23       max_time: 20m
24     experiment:
25       max_failed_trials: 10%
26       workloads:
27         workload_a:
28           confidence_interval_metric: avg_response_time
29           confidence_interval_value: 300ms
30           confidence_interval_precision: 90%
31   quality_gates:
32     workloads:
33       workload_a:
34         - max_mix_deviation: 5%
35           max_think_time_deviation: 2%
36           gate_metric: avg_response_time
```

```
37            condition: "<="
38            gate_threshold_target: "150ms"
39            gate_threshold_minimum: "300ms"
40  sut:
41    ...
42  workloads:
43    ...
44  data_collection:
45    ...
```

Listing 5.33. DSL: A Spike Test for a Web service YAML Example

In Listing 5.33 we report an example of a spike test goal. The spike test in the example states:

a) the goal as being a spike test (line #7);

b) the metrics to `observe` of interest for the scalability test (lines #8-#15), both on workloads and resource utilization;

c) the load function defined in such a way a spike in the workload is generated (lines #16-#20). The number of simulated users is 5000, expecting this number to generate a spike in the workload handled by the SUT. The `ramp_up` is of 1 minute in which the SUT is incrementally loaded up to 5000 simulated users interacting with it. The `steady_state` is rather low because the user wants to generate a spike, thus the `ramp_down` is defined in such a way the number of users is scaled down at the same pace it has been scaled up during the `ramp_up`;

d) the termination criteria, similar to the one defined for the scalability test in Listing 5.32, but for the spike test the accepted `confidence_interval_value` is higher and the `confidence_interval_precision` is lower compared to the mentioned test, because the load is much higher (lines #21-#30);

e) as per the termination criteria, also the quality gates are similar to the ones in Listing 5.32, but with updated values according to the higher load (lines #31-#39).

```yaml
version: "3"
name: "Exhaustive Exploration Test"
description: "Example of Exhaustive Exploration Test re-using Store Knowledge"
labels: "exhaustive_exploration", "stored_knowledge"
configuration:
  goal:
    type: "exhaustive_exploration"
    stored_knowledge: "true"
    observe:
      ...
    exploration:
      exploration_space:
        services:
          service_a:
            resources:
              cpu:
                range: [100m, 1000m]
                step: "^2"
              memory:
                range: [1024Mi, 256Mi]
                step: "/2"
            configuration:
              NUM_SERVICE_THREAD: [12, 24]
          dbms_a:
            resources:
              cpu:
                range: [100m, 1000m]
                step: "*4"
              memory:
                range: [256Mi, 1024Mi]
                step: "+512Mi"
            configuration:
              QUERY_CACHE_SIZE: [48Mi, 64Mi, 88Mi, 112Mi]
      exploration_strategy:
        selection: "one_at_a_time"
  load_function:
    users: 1000
    ramp_up: 5m
    steady_state: 20m
    ramp_down: 5m
  termination_criteria:
    ...
  quality_gates:
    ...
sut:
```

```
46    ...
47  workloads:
48    ...
49  data_collection:
50    ...
```

Listing 5.34. DSL: An Exhaustive Exploration Test for a Web service YAML Example

In Listing 5.34 we report an example of exhaustive exploration test goal. The user's intent could be to learn about the performance of the developed system when setting different configurations and resource allocations. The exhaustive exploration test in the example states:

a) the goal as being an exhaustive exploration test (line #7);

b) the reuse of `stored_knowledge`, given the complexity of the exploration test (line #8);

c) the exploration space (lines #12-#33) and the exploration strategy (lines #34-#35);

d) the load function at lines #36-#40.

Exploration test specifications are similar to the ones for configuration tests. It is common to reuse specifications for the mentioned types of tests.

```
1   version: "3"
2   name: "Stability Boundary Test"
3   description: "Example of Stability Boundary Test"
4   labels: "stability_boundary", "stored_knowledge"
5   configuration:
6     goal:
7       type: "stability_boundary"
8       stored_knowledge: "true"
9       observe:
10        workloads:
11          workload_a: avg_response_time, avg_latency
12          workload_b: avg_response_time, avg_latency
13          workload_b.operation_a: avg_response_time
14        services:
15          service_a: avg_cpu, avg_memory
```

```
        service_b: avg_cpu, avg_memory
        dbms_a: avg_cpu, avg_memory, avg_io
    exploration:
      exploration_space:
        services:
          service_a:
            resources:
              cpu:
                range: [100m, 1000m]
                step: "*4"
              memory:
                range: [256Mi, 1024Mi]
                step: "+768Mi"
            configuration:
              NUM_SERVICE_THREAD: [12, 24]
          dbms_a:
            resources:
              cpu:
                range: [100m, 1000m]
                step: "*10"
              memory:
                range: [256Mi, 1024Mi]
                step: "+768Mi"
            configuration:
              QUERY_CACHE_SIZE: 48Mi
        stability_criteria:
          services:
            service_a:
              avg_cpu: "<=60%"
              avg_memory: "<=80%"
            dbms_a:
              avg_memory: "<=70%"
          workloads:
            workload_b:
              max_mix_deviation: 5%
      exploration_strategy:
        selection: "stability_boundary_first"
  load_function:
    users: 1000
    ramp_up: 5m
    steady_state: 20m
    ramp_down: 5m
  termination_criteria:
    ...
  quality_gates:
    ...
```

```
sut:
  ...
workloads:
  ...
data_collection:
  ...
```

Listing 5.35. DSL: A Stability Boundary Test for a Web service YAML Example

In Listing 5.35 we report an example of stability boundary test goal. The specification is similar to the one in Listing 5.14, and report all the entities needed to specify a stability boundary test.

```
version: "3"
name: "Capacity Test"
description: "Example of Capacity Test with constraints"
labels: "capacity", "constraints"
configuration:
  goal:
    type: "capacity_constraints"
    observe:
      workloads:
        workload_a: avg_response_time, avg_latency
        workload_b: avg_response_time, avg_latency
      services:
        service_a: avg_cpu, avg_memory
        service_b: avg_cpu, avg_memory
        dbms_a: avg_cpu, avg_memory, avg_io
    exploration:
      exploration_space:
        load_function:
          users:
            range: [250, 5000]
            step: "+250"
      exploration_strategy:
        selection: "one_at_a_time"
  load_function:
    ramp_up: 10m
    steady_state: 60m
    ramp_down: 10m
  termination_criteria:
    test:
      max_time: 72h
```

```
      max_failed_experiments: 0%
    experiment:
      max_failed_trials: 10%
      workloads:
        workload_a:
          confidence_interval_metric: avg_response_time
          confidence_interval_value: 200ms
          confidence_interval_precision: 95%
  quality_gates:
    workloads:
      workload_a:
        - max_mix_deviation: 5%
          max_think_time_deviation: 2%
          gate_metric: avg_response_time
          condition: "<="
          gate_threshold_target: "100ms"
          gate_threshold_minimum: "200ms"
    services:
      service_a:
        - gate_metric: avg_cpu
          condition: ">="
          gate_threshold_target: 20%
          gate_threshold_minimum: 10%
        - gate_metric: avg_cpu
          condition: "<="
          gate_threshold_target: 50%
          gate_threshold_minimum: 60%
        - gate_metric: avg_memory
          condition: ">="
          gate_threshold_target: 256Mi
          gate_threshold_minimum: 128Mi
        - gate_metric: avg_memory
          condition: "<="
          gate_threshold_target: 512Mi
          gate_threshold_minimum: 512Mi
      service_b:
        - gate_metric: avg_cpu
          condition: ">="
          gate_threshold_target: 20%
          gate_threshold_minimum: 10%
        - gate_metric: avg_cpu
          condition: "<="
          gate_threshold_target: 50%
          gate_threshold_minimum: 60%
        - gate_metric: avg_memory
          condition: ">="
```

```
          gate_threshold_target: 128Mi
          gate_threshold_minimum: 64Mi
        - gate_metric: avg_memory
          condition: "<="
          gate_threshold_target: 256Mi
          gate_threshold_minimum: 256Mi
      dbms_a:
        - gate_metric: avg_cpu
          condition: ">="
          gate_threshold_target: 20%
          gate_threshold_minimum: 10%
        - gate_metric: avg_cpu
          condition: "<="
          gate_threshold_target: 50%
          gate_threshold_minimum: 60%
        - gate_metric: avg_memory
          condition: ">="
          gate_threshold_target: 256Mi
          gate_threshold_minimum: 128Mi
        - gate_metric: avg_memory
          condition: "<="
          gate_threshold_target: 512Mi
          gate_threshold_minimum: 512Mi
sut:
  name: "my_app"
  versions:
    values: "v1.5"
  type: "http"
  sut_configuration:
    ...
  services_configuration:
    service_a:
      resources:
        cpu: 500m
        memory: 512Mi
      configuration:
        THREADPOOL_SIZE: 64
    service_b:
      resources:
        cpu: 200m
        memory: 256Mi
      configuration:
        THREADPOOL_SIZE: 64
    dbms_a:
      resources:
        cpu: 800m
```

```
123         memory: 512Mi
124       configuration:
125         QUERY_CACHE_SIZE: 88Mi
126 workloads:
127   ...
128 data_collection:
129   ...
```

Listing 5.36. DSL: A Capacity Constraints Test for a Web service YAML Example

In Listing 5.36 we report an example of a capacity test with constraints goal. The capacity test with constraints in the example states:

a) the goal as being a capacity constraints test (line #7);

b) the metrics to `observe` of interest for the capacity test (lines #8-#15), both on workloads and resource utilization;

c) the exploration space defined on the number of users, from the lower to the higher number of users (lines #17-#21). By increasing the number of users with the defined step (line #21), the user tests for the capacity of the system;

d) the exploration strategy at lines #22-#23;

e) the load function at lines #24-#27 lasting longer than other kinds of test, because more time is required to assess the capacity of the SUT under a sustained load;

f) the termination criteria a lines #28-#38. In the case of a configuration test, the user usually does not accept failing experiments as specified in line #31. This is because the capacity test is not expected to have failing experiments, and if there are, the result shall not be considered valid, thus the test can be terminated right away;

g) the quality gates for the capacity test, defining the actual constraints accepted for the SUT to be considered successfully handling the issued load. The quality gates check for different resource-related metrics on all the services part of the SUT (lines #48-#99), to ensure resource utilization is on average under defined quality criteria representing the constraints of the capacity test. The quality gates in this example check

both for a lower threshold in resource utilization (e.g., lines #50-#53), as well as an upper threshold (e.g., lines #54-#57). This is important because we do not want the SUT to be either under or over-utilizing the allocated resources in the range of users we use to test the SUT capacity;

h) the `sut` configuration for the services part of the SUT. For a configuration test, it is important to define a given configuration for all the services, which is the configuration of the SUT for which the capacity is tested.

The configuration of the services part of the SUT is usually defined using for example an exploration test.

```yaml
version: "3"
name: "Acceptance Test"
description: "Example of Acceptance Test"
labels: "acceptance_test"
configuration:
  goal:
    type: "acceptance_test"
    # stored_knowledge: "false"
    observe:
      workloads:
        workload_a: avg_response_time
        workload_b: avg_response_time
  load_function:
    users: 1000
    ramp_up: 5m
    steady_state: 20m
    ramp_down: 5m
  termination_criteria:
    test:
      max_time: 50m
    experiment:
      max_failed_trials: 0%
      workloads:
        workload_a:
          confidence_interval_metric: avg_response_time
          confidence_interval_value: 200ms
          confidence_interval_precision: 95%
  quality_gates:
    workloads:
      workload_a:
        - max_mix_deviation: 5%
          max_think_time_deviation: 2%
          gate_metric: avg_response_time
```

```
34            condition: "<="
35            gate_threshold_target: "100ms"
36            gate_threshold_minimum: "200ms"
37  sut:
38    ...
39  workloads:
40    ...
41  data_collection:
42    ...
```

Listing 5.37. DSL: An Acceptance Test for a Web service YAML Example

In Listing 5.37 we report an example of an acceptance test goal. The acceptance test in the example states:

a) the goal as being an acceptance test (line #7);

b) the metrics of interest to be evaluated as part of the acceptance criteria (lines #9-#12). For an acceptance test the metrics of interest usually are few and corresponds to important performance metrics for the SUT;

c) the load function at lines #13-#17;

d) the termination criteria at lines #18-#27;

e) the quality gates at lines #28-#36 representing the acceptance criteria for the acceptance test.

The example in Listing 5.37 is very similar to the load test example in Listing 5.27. Many tests share a similar structure along with the entire test definition or parts of the same. As for example, many tests might share the `observe`d metrics, the `workloads`, and `data_collection` services. In these cases the user can define a test, facilitated by the native inheritance mechanism provided by the YAML language, as well as the overriding mechanism offered by the framework part of the proposed approach. This is something we allow as part of the proposed solutions, to facilitate reusability of basic tests and opinionated templates defined by expert users and/or provided as part of our approach. A user could define basic tests, including all the details about the workload, the SUT, collections services, etc and then override such specifications with an additional specification providing only the delta to be applied besides (if not present in the basic specification) or in substitution (if present in the basic specification) information.

```yaml
version: "3"
name: "Regression Test"
description: "Example of Regression Test"
labels: "regression_complete"
configuration:
  goal:
    type: "regression_complete" # OR regression_intersection
    stored_knowledge: "true"
    observe:
      workloads:
        workload_a: avg_response_time
        workload_b: avg_response_time
  load_function:
    users: 1000
    ramp_up: 5m
    steady_state: 20m
    ramp_down: 5m
  termination_criteria:
    test:
      max_time: 3h
    experiment:
      max_failed_trials: 0%
      workloads:
        workload_a:
          confidence_interval_metric: avg_response_time
          confidence_interval_value: 200ms
          confidence_interval_precision: 95%
  quality_gates:
    regression:
      workload: workload_a
      gate_metric: avg_response_time
      regression_delta_absolute: 50ms
    workloads:
      workload_a:
        - max_mix_deviation: 5%
          max_think_time_deviation: 2%
          gate_metric: avg_response_time
          condition: "<="
          gate_threshold_target: "150ms"
          gate_threshold_minimum: "250ms"
sut:
  name: "my_app"
  versions:
    values: [ "v1.5", "v1.6" ]
  type: "http"
```

```
46    sut_configuration:
47      ...
48    services_configuration:
49      service_a:
50        resources:
51          cpu: 500m
52          memory: 512Mi
53        configuration:
54          THREADPOOL_SIZE: 64
55      service_b:
56        resources:
57          cpu: 200m
58          memory: 256Mi
59        configuration:
60          THREADPOOL_SIZE: 64
61      dbms_a:
62        resources:
63          cpu: 800m
64          memory: 512Mi
65        configuration:
66          QUERY_CACHE_SIZE: 88Mi
67  workloads:
68    ...
69  data_collection:
70    ...
```

Listing 5.38. DSL: A Regression Test for a Web service YAML Example

In Listing 5.38 we report an example of a regression test goal. The regression test in the example states:

a) the goal as being a regression test on the entire workload for all the versions of the SUT (`regression_complete` at line #7);

b) the reuse of `stored_knowledge`, so that access to previous regression tests result is granted (line #8);

c) the metrics of interest for the regression (lines #9-#12);

d) the load function at lines #13-#17;

e) the termination criteria at lines #18-#27;

f) the quality gates at lines #28-#40, defining the regression criteria (lines #29-#32) and a threshold for the `avg_response_time` of `workload_a`, the same workload of interest of the regression criteria;

g) the SUT configuration specifying the versions of interest for the regression (lines #43-#44) and the configuration of the services (lines #48-#66) so that the same configuration is used for the regression evaluation.

DSL for CSDL YAML Examples

In this section, we present some examples of test suites specifications. After specifying different tests, users can also define test suites to integrate subsets of tests as part of the CSDL. To do so, users are expected to define:

a) The test suite, selecting tests by referring to YAML files defining them or by selecting them using labels;

b) The triggers activating the test execution, according to CSDL events;

c) Optionally, the environments on which tests have to be executed;

d) The expected outcome of the test suite according to defined quality criteria.

```yaml
version: "1.6"
name: "Smoke Tests on Features"
description: "Runs all the Smoke Tests on new features merging to the Development
↪  branch"
suite:
  triggers:
    on:
      pull_request:
        contexts: "merge"
        source_branches:
          - "feat-*"
        target_branches:
          - "development"
  tests:
    include_labels: [ "(.*)smoke_test(.*)" ]

quality_gates:
  criterion: "all_success"
```

```
18    exclude:
19      - "./tests/performance/smoke/smoke_test_alternative_scenario.yaml"
```

Listing 5.39. DSL for CSDL: A Smoke Tests Suite YAML Example

In Listing 5.39 we report an example of a test suite specification executing smoke tests. The test suite in the example states:

a) the tests included in the suite are the ones with labels matching the regular expression `(.*)smoke_test(.*)` (lines #13-#14);

b) the trigger of the test suite is a pull request coming from a branch matching the regular expression `feat-*` and targeting the `development` branch. The tests are executed on the SUT built out of the `feat-*` branch code base merged with the codebase in the `development` branch (lines #5-#12);

c) the test suite is considered successful when all the executed tests, but the one indicated at line #19, are successful, i.e., each test passes the quality gates specified for the given test (line #17).

```
1   version: "1.6"
2   name: "Acceptance Tests on Features"
3   description: "Runs all the Acceptance Tests on Pushes to Hotfix branches"
4   suite:
5     triggers:
6       on:
7         push:
8           branches:
9             - "hotfix-*"
10    tests:
11      include_labels: [ "(.*)acceptance_test(.*)" ]
12
13  quality_gates:
14    criterion: "all_success"
```

Listing 5.40. DSL for CSDL: An Acceptance Tests Suite YAML Example

In Listing 5.40 we report an example of test suite specification executing acceptance tests. In this example, the user is defining a test suite to be executed on hotfixes. The test suite in the example states:

a) the tests included in the suite are the ones with labels matching the regular expression `(.*)acceptance_test(.*)` (lines #10-#11);

b) the trigger of the test suite is a push to branches matching the regular expression `hotfix-*` (lines #7-#9);

c) the test suite is considered successful when all the executed tests are successful, i.e., each test passes the quality gates specified for the given test ((line #14).

```
1  version: "1.6"
2  name: "Regression Tests Suite"
3  description: "Runs all the Regression Tests on a Scheduled basis"
4  suite:
5    triggers:
6      scheduled: true
7    tests:
8      include_labels: [ "(.*)regression_test(.*)" ]
9
10 quality_gates:
11   criterion: "all_success"
```

Listing 5.41. DSL for CSDL: A Regression Tests Suite YAML Example

In Listing 5.41 we report an example of test suite specification executing regression tests. The test suite in the example states:

a) the tests included in the suite are the ones with labels matching the regular expression `(.*)regression_test(.*)` (lines #7-#8);

b) the test suite is scheduled to be executed by the CIS or the CDS tool used in the CSDL of the SUT (lines #5-#6). For example, regression tests are usually scheduled in different moments of the day;

c) the test suite is considered successful when all the executed tests are successful, i.e., each test passes the quality gates specified for the given test (line #10).

5.6.4 The Scala-Java Library

The complexity and expressiveness of the presented models require to be properly codified to empower it successfully, both from the user perspective, as well

as from the framework part of the provided approach. Moreover, we anticipate a successful utilization of the approach, IDE integration is going to be very important. To achieve this, we implemented the model in a software library, mostly realized using Java and Scala programming languages. In Chap. 6 we provide more detail about how the library is embedded as part of the proposed approach and how it is exposed to the users. In this section we highlight two important features embedded in the implemented library: a) syntactic and semantics validation of the model; b) generation of the exploration space.
The library is built to evolve with the DSL versions. It is built to be extended, ensuring the consistency of the model in terms of syntax and semantics.

Syntactic and Semantics Validation

The user submits test execution specifications relying on test bundles. By leveraging the model presented in Sect. 5.4, the library ensures the submitted specifications are syntactically and semantically valid before a test can be instantiated. This is very important in the context of performance testing and CSDL because performance tests usually require a fair amount of time to be executed. So everything that can be verified statically must be verified statically, and erroneous test definitions must be spotted early. The DSL, CSDL and SUT deployment descriptor model definitions support by design syntactic validation, by ensuring entities can be parsed only if correctly structured and only if using the data types we enforce in the model. For what concern semantics validation, we ensure the definition of the test, the test suite, and the SUT deployment descriptor are consistent, by verifying every single entity is defined in a semantically correct way (e.g., data collectors requiring configurations are specified if some metrics computed on top of performance data collected by those collectors are declared as observed by the user), and that the entire definition has not conflicting statements (e.g., we assert that if a test defines an exploration on the load function, then the same is not also defined as part of the configuration). We also make sure that according to the test goal type specified by the user, the test is consistent and complete in terms of the specified entities. We also ensure semantics correctness of the test specification for the following language elements:

a) when data collection services are specified, the user is expected to specify observed metrics;

b) if no observed metrics nor data collection services are specified, the library adds default observed metrics and data collection services, including both

client-side performance metrics on all the workloads (i.e., response time, latency, throughput) and server-side performance metrics on all the services (i.e., `CPU`, `RAM`, `DISK`, and `NET` utilization). In this case, the test is scheduled and the user is informed;

c) we ensure the `max_time` and the `max_number_of_experiments` specification are consistent with the load function and the amount of experiments to be executed;

d) we ensure, when `operations` declare dependencies to `data_sources` for retrieving data, that the referenced files are provided as part of the `test bundle` and are parsable according to the specification in the `data_source`. We do the same when `operations` refer to BPMN 2.0 models for `wfms`s;

e) we ensure the `mix` specification is consistent with the `operations` specification;

f) we ensure the popularity specification for both the `workload` and the `workload item` is consistent and for each entity sums up to 100%, after automatically adding the omitted `popularity` fields;

g) we ensure that, when a predictive model is specified as `exploration_strategy`, the `mean_absolute_error` is specified as part of the quality gates;

h) we ensure the workload specification is consistent with the resources available on the test infrastructure for load drivers. For example, we ensure the test infrastructure can simulate the requested amount of users.

For the semantics correctness of the test suite specification we ensure:

a) the test suite specifies at least one trigger;

b) the test suite selection criteria include at least one test;

c) the *Quality Gates* are applied at least to one test, after excluding the tests specified as `exclude`.

For the semantics correctness of the SUT deployment descriptor specification we make sure that when the specification of the test includes the deployment of the SUT, we validate that the provided and generated deployment descriptors are consistent in terms of resource demand (for `CPU`, `RAM`, `DISK`, and `NET`)

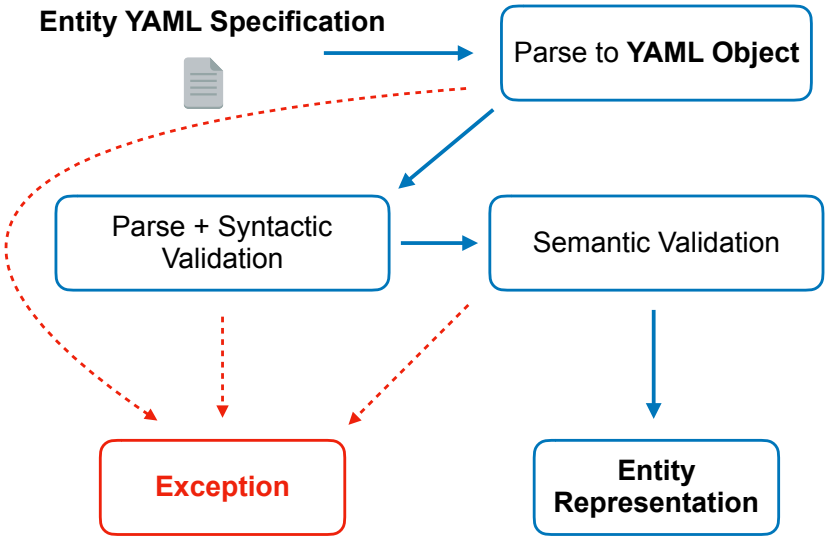

Figure 5.33. The DSL Parsing and Validation Process

according to the resources available on the deployment infrastructure for the test.

The semantics checks mentioned above, are important to guarantee the user is scheduling a test or test suite that can fail only in case something unexpected happens during the execution at runtime, e.g., the SUT fails the deployment or has an error in handling the issued load. If syntactic and/or semantics checks fail, the user is warned immediately so she has the opportunity to improve the test, test suite or SUT deployment descriptor specifications before submitting it for execution.

In Fig. 5.33 we report the parsing and validation process. The syntactic checks are carried out first and enable the library to validate that the types are corresponding to the meta-model in Fig. 5.1. The implementation we provide in this thesis is based on the work in [D'Avico, 2016] and [Findahl, 2017]. More details on the actual technical implementation can be found on the mentioned theses. These checks are part of the parsing from the YAML text format into the DSL, or CSDL, or SUT deployment descriptor according to the actual specification getting parsed. The first step of the parsing is to convert the string representation into an Abstract Syntax Tree (AST) or `YAML Object`. When more than one YAML specification for a single meta-model is provided, e.g., because the user is overriding specifications for reusability, the parsed specifications are merged after the parsing to an AST and the result is used for the next steps. When merging the specification, and especially when overrides are involved, the AST

ensures consistency and correctness. The exception messages generated when reading a YAML file always contain the `key` where an error is detected, referring to the actual entity in the meta-model where the error is present. For instance if the user forgets to specify a `goal type` or the `goal type` is not known to the system, the user would get a message containing *"configuration.goal.type"* to easily trace back the error in the YAML file. After ensuring the specifications can be correctly parsed according to the YAML standard, we parse the YAML specification to a Java and Scala representation of the corresponding meta-model and ensure the syntactic of the same is consistent. After ensuring the specification is syntactically correct, we apply the semantic checks to all the entities, to guarantee the specification is consistent according to the semantics of the meta-model and the resources available on the test infrastructure. When all the steps in the parsing process are complete, the YAML entities specifications are parsed into Java and Scala entity representations. If on the other end, some exceptions are thrown during the parsing process we report this back to the user and stop the execution of any further steps. For what concern experiments specification and generated SUT deployment descriptor models, e.g., when exploration spaces are involved, a similar process is followed when they are parsed. The difference is that the specifications for the mentioned entities are generated out of the Java and Scala model so they are validated by design and no exception is expected when parsed by the library.

Computation of Exploration Space

Some of the supported test goals, require the computation of a performance exploration space according to provided specifications. We support the generation of the exploration space as part of the implemented library. This allows us to generate all the experiments part of the exploration space automatically, and ensure they are consistent with the specification and supported by the underlying test infrastructure. The exploration space could explode in terms of the number of experiments needing to be scheduled, and consequently in terms of time. To help the user in better understanding the time and space requirements of the specified test and test suite, we:

a) warn the user if the total execution time of the test or the test suite is above 4 hours according to the number of experiment and trials, and the load function specification;

b) warn the user if the number of specified experiments in the exploration space exceeds 10.

The actual thresholds mentioned in the previous checks represent the default ones. They can be customized by the user as part of the configuration of the framework part of the proposed approach. With these two checks and consequent warnings to the user, we ensure the user is scheduling the test knowing the actual duration time and the number of experiments. We report back to the user, along with the warning, the actual expected minimum duration time, and the number of experiments part of the test, as well as an indication of the expected number of trials. If the user decides to schedule the test anyway, we ensure the actual number of experiments in the exploration space can be specified and held in Java and Scala programming languages types.
As per the specification, we generate the exploration space by computing the Cartesian product without repetitions of all the dimensions and the values assigned to those dimensions, specified by the user in the `exploration_space` section of the test specification. The implementation in this thesis is based on the work in [Findahl, 2017]. More details on the actual technical implementation can be found in the mentioned thesis.

```
1 case class ExplorationSpace(
2     size: Int,
3     usersDimension: Option[List[NumUsers]],
4     cpuDimension: Option[Map[ServiceName, List[Millicores]]],
5     memoryDimension: Option[Map[ServiceName, List[Bytes]]],
6     configurationDimension: Option[Map[ServiceName, Map[VariableName,
      ↪  List[VariableValue]]]]
7 )
```

Listing 5.42. Exploration Space Data Structure

Listing 5.42 depicts the `ExplorationSpace` data structure, where we store the exploration space dimensions. This data structure stores, for each variable, all the possible values the variable takes following the order of specification according to the model. For instance to get the configuration of the first exploration point the index (0) is used to access the value of the variable in each `List`. By using the `Option`al Scala type we can include or omit a variable while at the same time keeping the same overall data structure in all cases. In the `ExplorationSpace` we also store the `size` (e.g., the number of points in the exploration space) to enable easy access in various parts of the code (line #2).

```
case class ExplorationSpaceDimensions(
    users: Option[List[NumUsers]],
    cpu: Option[Map[ServiceName, List[Millicores]]],
    memory: Option[Map[ServiceName, List[Bytes]]],
    configuration: Option[Map[ServiceName, Map[VariableName,
    ↪  List[VariableValue]]]]
)
```

Listing 5.43. Exploration Space Dimensions Data Structure

To generate the `ExplorationSpace`, we use the Cartesian product, and we remove repetitions. Since the Cartesian product is deterministic it is possible to calculate each combination (i.e., exploration point configuration) variable by variable.

```
GENERATE-EXPLORATION-SPACE(ExplorationSpaceDimensions)
  explorationSpace = {}
  size = CALCULATE-SIZE(ExplorationSpaceDimensions)
  blockSize = size
  FOR each variable(v) in ExplorationSpaceDimensions
    blockSize = blockSize / v.values.length
    order = COMPUTE-ORDER-OF-VALUES(blockSize, v.values.length, size)
    explorationSpace[v] = REPLACE-WITH-VALUES(order, v.values)
  return (size, explorationSpace)

CALCULATE-SIZE(ExplorationSpaceDimensions)
  sizes = []
  FOR each variable(v) in ExplorationSpaceDimensions
    sizes.add(v.values.length)
  IF sizes.length == 0
    size = 0
  ELSE
    size = 1
    FOR each size(s) in sizes
      size = size * s
  return size

COMPUTE-ORDER-OF-VALUES(blockSize, numValues, listLength)
```

```
24  order = []
25  FOR i = 0 to listLength / (blockSize * numValues)
26    FOR value = 0 to numValues
27      order = order + FILL(blockSize, value)
28  return order
```

Listing 5.44. Exploration Space Generation Algorithm

Listing 5.44 describes the algorithm used for generating the exploration space in pseudo-code. The algorithm is applied to the data stored in the `ExplorationSpaceDimensions` data structure reported in Listing 5.43. `ExplorationSpaceDimensions` stores all the possible values for each dimension, as specified by the user in the test specification. First, we calculate the total size of the exploration space by multiplying the number of possible values for each variable (line #3 referencing to lines #11-#21). For each variable, we then compute the order of the values so that we get all possible combinations (line #7 referencing lines #23-#28). The `blocksize` is used to keep track of the number of repetitions of a specific value and is updated after each dimension. In `COMPUTE-ORDER-OF-VALUES` we use the `FILL` function to generate a list of the size `blockSize` that is filled with a given `value`. After the order is computed we replace the indices with the real values in the `REPLACE-WITH-VALUES` method (line #8). For instance if there are two values and the `blockSize` is 2 and the size of the exploration space is $2*2 = 4$, `COMPUTE-ORDER-OF-VALUES` would return `[0,0,1,1]`, and if the `blockSize` is 1 it would return `[0,1,0,1]`. Overall this yields the *Cartesian product* (i.e., all possible combinations of values without repetition): `[(0,0),(0,1),(1,0),(1,1)]`.
In Figure 5.34 we show a concrete example of the exploration space generation of $size = 6$, where two dimensions, users and memory, are specified. This algorithm has a complexity of $O(N)$ where N is the exploration space size. Although there might exist similar algorithms with lower complexity, the algorithm described here suffice because the computational time is a fraction of the time it takes to run the performance test itself. Although there is a theoretical possibility of a space explosion in practice the likelihood is minimal when we consider the domain it is used for, and the thresholds in place for the number of experiment and exploration space dimensions.
To extend the *Exploration Space* with additional variables it is necessary to add it to the test specification meta-model, the *ExplorationSpace*, and the *ExplorationSpaceDimensions* data structures, as well as to the flattening algorithm.

The following exploration space:

- `users = [500,1000,1500]`
- `memory = [256Mi,512Mi]` (for a given service)

yields to the following exploration space computation:

1. `size = 3*2 = 6`
2. `blockSize = size = 6 (updated after each step below)`
3. `usersDimension:`
   (a) `blockSize = blockSize/v.values.length = 6/3 = 2`
   (b) `order = [0,0,1,1,2,2]`
   (c) `explorationSpace[usersDimension] = [500,500,1000,1000,1500,1500]`
4. `memoryDimension:`
   (a) `blockSize = blockSize/v.values.length = 2/2 = 1`
   (b) `order = [0,1,0,1,0,1]`
   (c) `explorationSpace[memoryDimension] = [256Mi,512Mi,256Mi,512Mi,256Mi,512Mi]`
5. `explorationSpace = (usersDimension, memoryDimension) = [(500, 256Mi), (500, 512Mi), (1000, 256Mi), (1000, 512Mi), (1500, 256Mi), (1500, 512Mi)]`

Figure 5.34. A Concrete Exploration Space Generation Example

This allows us to have control in the extension of the exploration space, at the expense of a little overhead, to ensure the test specification is consistent.

5.7 Concluding Remarks

In this chapter, we present the declarative DSL we propose for performance test automation, and integration with the CSDL. We illustrate the meta-model and the model of the entities of the language, as well as the YAML serialization specification we propose as the representation of the model to the users and the library we develop to codify the DSL. We also present comprehensive examples of specifications for the different elements of the model, as well as for the different supported test goals. We also discuss how the provided DSL is used to validate the test specifications, and how we rely on the DSL to generate the entire set of experiments, and SUT deployment descriptors needed to achieve the goal specified as part of the test. The proposed DSL tackle R.G. 2 presented in Sect. 1.2.1. It allows users to declaratively specify performance tests and their automation processes codifying the users' goal and offering an abstraction layer for the users to specify all the data needed for the automated execution of performance tests out of the proposed catalog. The DSL is used as the input for the framework part of the proposed approach we present in Chap. 6, taking care of executing the performance tests according to the specification. The proposed DSL for CSDL tackle R.G. 4 presented in Sect. 1.2.1. It allows the integration of the proposed DSL for automated performance test execution, in the CSDL. We facilitate the integration, by providing declarative abstractions allowing to map CSDL events to test suites of tests defined using the proposed DSL. The library codifying the DSL allows for integration with other tools part of the CSDL, relying on the library to parse, serialize or generate performance tests.

Chapter 6

BenchFlow: A Framework for Declarative Performance Testing

6.1 Introduction and Requirements

In this chapter, we present the BenchFlow framework. The framework completes the proposed DPE approach for performance testing, by providing and managing the underlying infrastructure for test scheduling, execution, and performance metrics computation that can be programmed using the DSL proposed in Chap. 5. The framework is designed to completely assist the user in all the activities that need to be executed for automating performance tests execution, such as test scheduling, handling of load and SUT deployment infrastructure, deploying the SUT, issuing the workload, collecting client- and server-side data, undeploying the SUT and data analysis. The framework has been named BenchFlow after the BenchFlow project presented in Sect. 1.5.
Figure 6.1 present the high-level view of the BenchFlow model-driven framework, the different stages involved for a successful declarative performance testing automation, and the main input and output data of the different stages. The BenchFlow framework automates the end-to-end execution of performance tests, by executing all the activities that are needed to answer the performance intent of the user, declared as the goal of the performance test. The first stage is the *Exploration* stage. The exploration phase handles the way a performance test is executed to reach its goal, and it is the central phase of the lifecycle. In this stage, the users are expected to provide the test specifications, as well as test data and a SUT deployment descriptor as part of the so-called *test bundle*. When a test suite is involved, the user provides the *test suite bundle* as well. The *test bundle*, after being verified for syntactic and semantics correctness, is

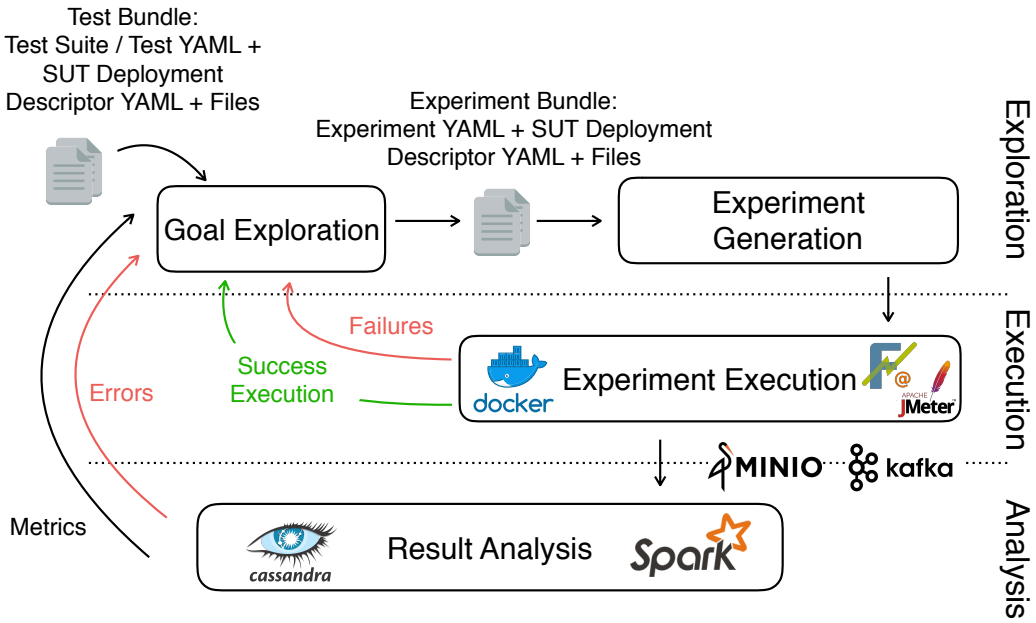

Figure 6.1. BenchFlow Framework: High-Level Diagram

used to generate the actual experiments to be executed, in a process referenced to as *Goal Exploration*. When a *test suite bundle* is submitted, it is also verified for syntactic and semantics correctness. The different *experiment bundles*, characterized by the experiment specification, a SUT deployment descriptor, and experiment data, are used as input for the *Experiment Generation*. Once experiments are generated, and ready to be scheduled on trial execution frameworks, the *Execution* stage is involved. In the execution stage, performance experiments and their different trials are executed relying on *Faban* or *JMeter* and *Docker*. Experiment execution can either be *successful* or *failing*, and the outcome is provided back to the *Goal Exploration* to be used for exploration decisions. At the end of each trial execution, performance data are collected and stored. We rely on *Minio* [MinIO, 2020], a bucket storage, to store raw performance data. After performance data collection, the *Analysis* stage is involved, and performance data is analyzed to compute performance metrics. The analysis is started according to collected performance data signaled in a work queue held by *Kafka* [Apache Software Foundation, 2020b]. Processed performance data are stored on *Cassandra* [Apache Software Foundation, 2020a], a NO-SQL DBMS, for efficiency in manipulating the data for the different computations. Actual performance data processing and metrics computation is performed relying on *Apache Spark* [Apache Software Foundation, 2020c]. Performance

metrics, and potential errors incurred during performance data processing and metrics computation, are reported back to the *Goal Exploration* to be included in decisions on how to continue with the exploration.

6.2 BenchFlow Framework Architecture

Figure 6.2 reports a high-level diagram of the framework, mainly focusing on the *Execution* and *Analysis* phases. The *Exploration* phase is realized by the core BenchFlow framework services and is discussed in the following sections. The *Execution* phase represents the servers deploying the services of the trial execution frameworks on one side, and an example of possible SUT services on the other side. The two sets of services are deployed on different servers, to guarantee control and quality of performance testing results. For all the services we rely on Docker containers for the runtime deployment. On the performance test execution side, we deploy either *Faban* or *JMeter* load drivers, according to the trial execution framework required by the test specification. The load drivers interact with the services part of the SUT by relying on software *adapters* facilitating and abstracting the interactions. Examples of software adapters are the ones we define for abstracting away the interaction with BPMN 2.0 WfMSs and provide SUT-awareness in the DSL. The load drivers also take care of interacting with services part of the proposed framework, *monitor*ing the SUT state and performance, and *collect*ing performance data. Collected performance data is then stored on *Minio*, and transformed into a standardized format using *Data Mappers* and *Data Transformers*. After standardizing the data format, performance data *analyzers* take charge of computing performance metrics and Key Performance Indicator (KPI) relying on *Apache Spark* and *Cassandra*.

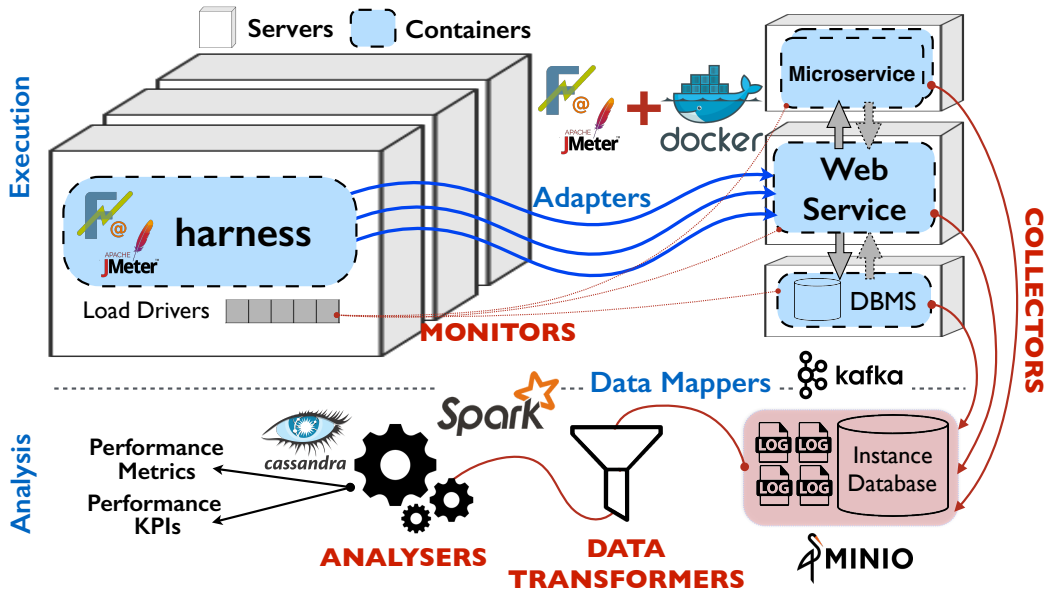

Figure 6.2. BenchFlow Framework: High-Level Architecture

6.2.1 Design Principles

The BenchFlow framework is designed according to the following principles:

1) model-driven and all the services and functionalities are configured and programmed via the DSL presented in Chap. 5;

2) functionalities are implemented in cohesive services scaling independently of each other, that can be deployed on a distributed infrastructure;

3) core functionalities and services implement state machines which state transitions are based on declarative test specifications and performance test results;

4) functionalities and services part of the framework are designed to facilitate reuse of features and performance test specifications;

5) functionalities are exposed via RESTful APIs, to facilitate human and third-party services integration and interactions;

6) monitor and collector services are implemented to be lightweight in terms of resource utilization so that they can be deployed next to the SUT services when required;

7) performance test execution is delegated to state-of-the-art tools, to guarantee the quality of the executed tests and the scalability of the load drivers infrastructure;

8) performance metrics, KPIs and statistical analysis of performance data is core to the framework;

9) performance data analysis and metric computation relies on a state-of-the-art framework to scale with the amount of data to be processed;

10) all the services are designed to facilitate extensions of functionalities when required;

11) the internal state of services, and all the generated and collected data, is accessible to users to guarantee traceability and reproducibility;

12) services are packaged in Docker images and deployed in Docker containers.

The rationale for leveraging Docker resides in the fact that containers do not add significant overhead to the system (assuming they are properly configured) [Bachiega et al., 2018], but bring several advantages to the table. More specifically, since Docker containers enable the control of all the configuration parameters of the SUT, repetitions (trials) of an experiment will always run from the same initial conditions, which are frozen in the corresponding Docker Image definitions. By relying on Docker, the reproducibility of the tests performed with BenchFlow is enhanced.

6.2.2 Components Diagrams

In this section we present the component diagrams, highlighting the main services part of BenchFlow and the principal interfaces among BenchFlow services and with third-party services. We then present the main services and their responsibility in the overall BenchFlow framework.
In Fig. 6.3 we present a component diagram of the main services part of BenchFlow, their main interfaces, and dependencies to other frameworks. The interfaces we present are mostly related to test scheduling and test execution. We divide the services to highlight which of them is related to each of the three phases introduced in Fig. 6.1. Related to the *Exploration* phase, we implement the following services:

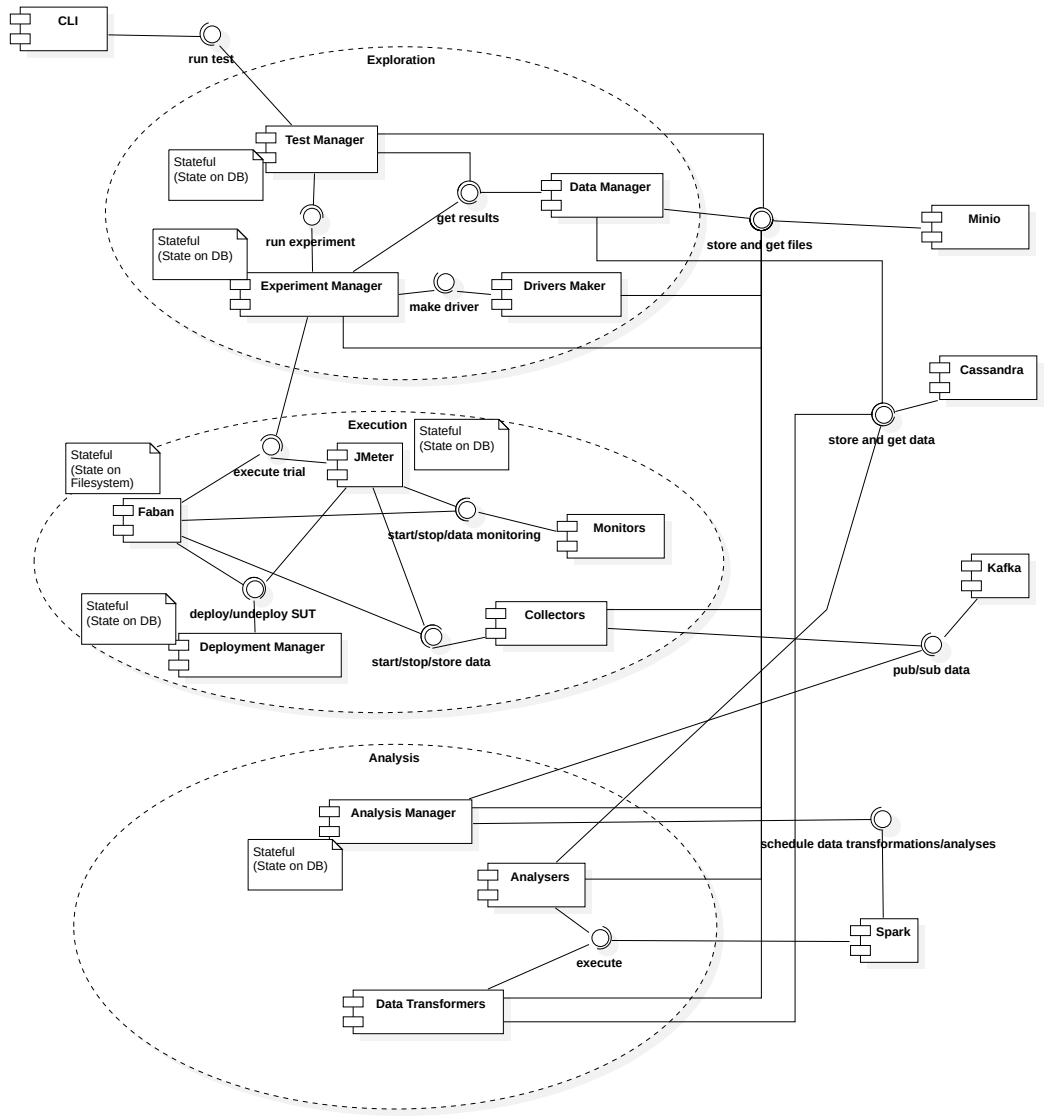

Figure 6.3. BenchFlow Framework: Services and Frameworks Component Diagram

a) the *test manager*, receiving the request to schedule a test or a test suite from the users, and handling the test execution;

b) the *experiment manager*, receiving the request to schedule an experiment from the *test manager*, and handling the experiment execution;

c) the *drivers maker*, generating the actual executable load drivers according to the trial execution framework and the experiment specification provided by the experiment manager;

d) the *data manager*, uniforming the access to performance metrics, KPIs, and statistics computed from the data collected from the SUT, providing dedicated APIs.

Users or other systems can rely on the *CLI* to interact with the test manager for submitting tests and test suites and accessing the state of test execution as well as results and metrics related to the same. When the test suite bundle is involved, the *Command-line Interface (CLI)* takes care of submitting all the test specifications related to the bundle, and keep track of all the test part of the test suite bundle. By doing so, we facilitate the integration with CSDL tools and allow the user to query for the state of the entire test suite, as well as enable the *CLI* to validate for the quality gates once all the involved tests complete the execution.
Related to the *Execution* phase, we implement the following services:

a) the *Faban drivers*, an extension of the standard Faban load drivers to inject BenchFlow automation facilities;

b) the *JMeter drivers*, an extension of the standard JMeter load drivers to inject BenchFlow automation facilities;

c) the *deployment manager*, responsible for deploying/undeploying the SUT, when requested by the Faban/JMeter drivers;

d) the *monitors*, services dedicated to monitoring the state of the SUT in different moments before, during and after the experiment execution, and controlled by the Faban/JMeter drivers;

e) the *collectors*, services dedicated to collect performance data from the SUT and the infrastructure underlying the SUT, and controlled by the Faban/JMeter drivers.

In the *Execution* phase we include the following third-party services:

a) the *Faban* framework, scheduling and executing Faban load drivers submitted by the experiments manager;

b) the *JMeter* framework, scheduling, and executing JMeter load drivers submitted by the experiments manager.

Related to the *Analysis* phase, we implement the following services:

a) the *analysis manager*, responsible for scheduling data processing on performance data collected during the test execution;

b) the *data transformers*, functions dedicated to transforming collected data in standard formats;

c) the *analyzers*, functions dedicated to analyze transformed data and compute performance metrics, performance KPIs, and statistics on them.

The BenchFlow services across all the phases rely on the following third-party services:

a) *Minio*, a bucket storage storing data collected from the SUT and the infrastructure underlying the SUT deployment, other than data related to the load drivers;

b) *Kafka*, a stream-processing software platform we rely on to signal for newly collected data to be processed;

c) *Spark*, a distributed general-purpose cluster-computing framework we rely on as runtime for data transformers and analyzers;

d) *Cassandra*, a NoSQL database management system designed to handle large amounts we rely on to store and read transformed performance data, performance metrics, KPIs, and statistics.

To facilitate the integration with the used third-party services we develop software libraries, the BenchFlow services interacting with the mentioned framework rely on.

Services and Responsibilities

In this section we detail services and responsibilities, along with enriching, compared to the previous section, the number of interfaces we present for each of the services, and the dependant services in the context of the same. Each diagram focuses on the interfaces exposed by the presented service, and then shows the main interfaces of other services the presented service calls.

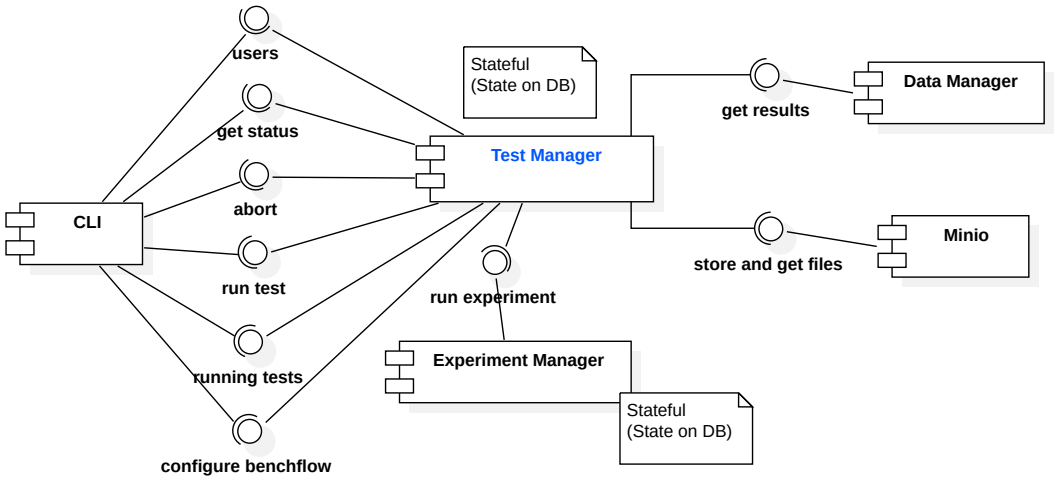

Figure 6.4. BenchFlow Framework: Test Manager Component Diagram

Exploration Phase **Test Manager** - In Fig. 6.4 we present the test manager. The test manager is a core service in BenchFlow and responsible for the *Goal Exploration*. Users of BenchFlow interact with the test manager to submit test and test suites and CSDL events data, as well as for obtaining the test execution status, the entire list of running tests, and aborting the test in case she decides to do so according to the obtained status. Submitted bundles are stored on *Minio* and available to other services needing to access the submitted declarative specifications. The test managers also offer a simplified way to support multiple users interacting with BenchFlow. We do not implement complex user management, but provides the possibility, via the CLI, to submit tests and interact with the test manager by specifying a custom user name. This way each user of the framework can select a different user name and we can warn the user if the user name has already been utilized. The test manager allows the users to obtain the entire list of users specified in the framework as well. As discussed in Chap. 5, the framework can be configured by the users. This can be done via the test manager as well, where the user can submit a configuration file used to configure BenchFlow settings.

```
testbed:
  sut:
      - { String : <name of a server> }:
        # OPTIONAL
        public_ips: { [String] : <a list of public ips> }
        private_ips: { [String] : <a list of private ips> }
```

```
7         # OPTIONAL
8         available_public_ports: { [Number] : <a list of numbers representing a
          ↪ port> }
9         available_private_ports: { [Number] : <a list of numbers representing a
          ↪ port> }
10      ...
11    load_drivers:
12      - { String : <name of a server> }:
13        # OPTIONAL
14        public_ips: { [String] : <a list of public ips> }
15        private_ips: { [String] : <a list of private ips> }
16        # OPTIONAL
17        available_public_ports: { [Number] : <a list of numbers representing a
          ↪ port> }
18        available_private_ports: { [Number] : <a list of numbers representing a
          ↪ port> }
19      ...
20  thresholds:
21    test_max_time: { String : <amount of time><unit> }
22    max_number_of_experiments: { String : <amount of time><unit> }
```

Listing 6.1. BenchFlow Framework: The Configuration File

In Listing 6.1 we present the configuration file format. The configuration file follows the YAML standard, and relies on the same data types of the DSL proposed in Chap. 5, and allows to configure:

a) the list and names of the servers where the SUT can be deployed to. Each server is represented by a unique name, a list of privates IPs and ports, and optionally a list of public IPs and ports (lines #2-#9);

b) the list and names of the servers where the load drivers can be deployed to. Each server is represented by a unique name, a list of privates IPs and ports, and optionally a list of public IPs and ports (lines #11-#18);

c) the default thresholds to be used for warning the user about the total execution time of the test (line #21), as well as the total number of experiments to be executed to explore the exploration space (line #22).

The interaction is facilitated by a **CLI** we provide as part of BenchFlow. The CLI abstracts the interactions with the test manager APIs and provides facilities to package and submit the test bundle. The CLI can also be used to

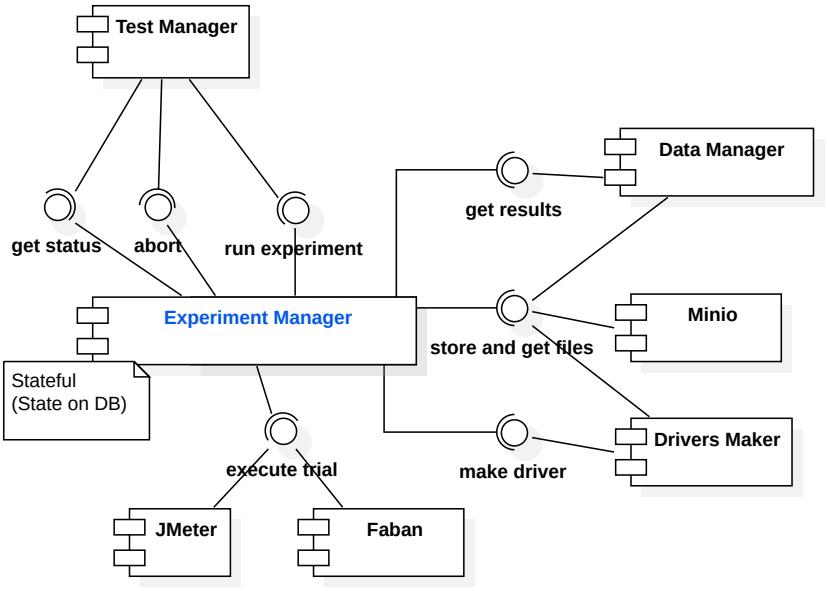

Figure 6.5. BenchFlow Framework: Experiment Manager Component Diagram

validate the test/test suite bundle, by requesting the test manager to check for syntactic and semantic validity of the same, without actually scheduling the section. This is useful during the definition of the specification, or for integration with IDEs providing syntax validation. The different interfaces also accept parameters, for example, to increase the verbosity of the response obtained by the user and the details included in the response. One example is when the user requests to access the status of the execution of a test. The simple response reports the running state, but the user can also specify parameters to obtain test results, and data on the entire exploration space part of the test, if any. During the *Goal Exploration*, the test manager needs to access data about the results of experiments and trials part of the test execution. To do so the test manager relies on the **Data Manager**, offering uniform access to the trials, experiments, and test results. The *Data Manager* is also enquired when the user state she wants to rely on *stored_knowledge* for the submitted test execution. The test manager, relying on the DSL library, generates the experiments specification as well as the SUT deployment descriptors for the same specification and builds the experiment bundles, storing them on *Minio*. The actual execution of experiments is then delegated to the *Experiment Manager*.
Experiment Manager - In Fig. 6.5 we illustrate the experiment manager and its main interfaces. The experiment manager is a core service and is responsible for executing the experiments on behalf of the test manager, defined according

to the provided declarative specification. The test manager, while exploring the goal, determines all the experiments to be executed, and requests the experiment manager to manage the execution. The experiment manager accesses the experiment bundles, retrieving them from *Minio*, and interacts with the *Drivers Maker*, a core service, to obtain the actual executable load drivers according to the target trial execution framework. The generated load drivers are stored on *Minio* and include all the data needed for automating the experiment execution on the target framework, i.e., the actual load driver, its configuration, the test data, and the SUT deployment descriptors for all the trials to be executed as part of the experiments, as well as the deployment descriptors for monitors and collectors. During the experiment execution, the experiment manager access trials' results relying on the *Data Manager*, another core service of the BenchFlow framework. The experiment manager relies on such a result to decide about the execution of the experiments. Other than managing the experiment execution, the experiment manager also offers interfaces to get the status of the running experiments and abort the experiment execution. These interfaces are called by the test manager, when the corresponding APIs of the test manager are interrogated by the users, or when required by the *Goal Exploration*. Through the *get status* interface of the experiment manager, the status of trials can be obtained as well. The actual execution of performance tests, and in particular of trials part of the experiment execution, is delegated to *Faban* or *JMeter*.

Execution Phase **Faban** and **JMeter** - In Fig. 6.6 we present the two frameworks we rely on for executing the experiment trials. We opted for Faban because is a solid framework used in performance, scalability and load testing of almost any type of server application and it offers mechanism and facilities for ensuring the load drivers execution is successfully managed. JMeter, on the other end, is widely used for real-world performance testing, and offer richer support for workload specification and SUT protocols compared to Faban. To facilitate the integration with the BenchFlow services, we develop a library for both of the frameworks, uniforming the interfaces for executing, aborting, and get the status (and the logs) of a trial. We also implement abstractions on top of the standards **Faban Drivers** and **JMeter Drivers**, to facilitate declarative performance testing automation. The provided abstractions, allow us to integrate with the load drivers lifecycle of both of the frameworks, facilities to handle SUT deployment and undeployment, controlling *Monitors* and *Collectors*, as well as managing known unexpected behavior during the trial ex-

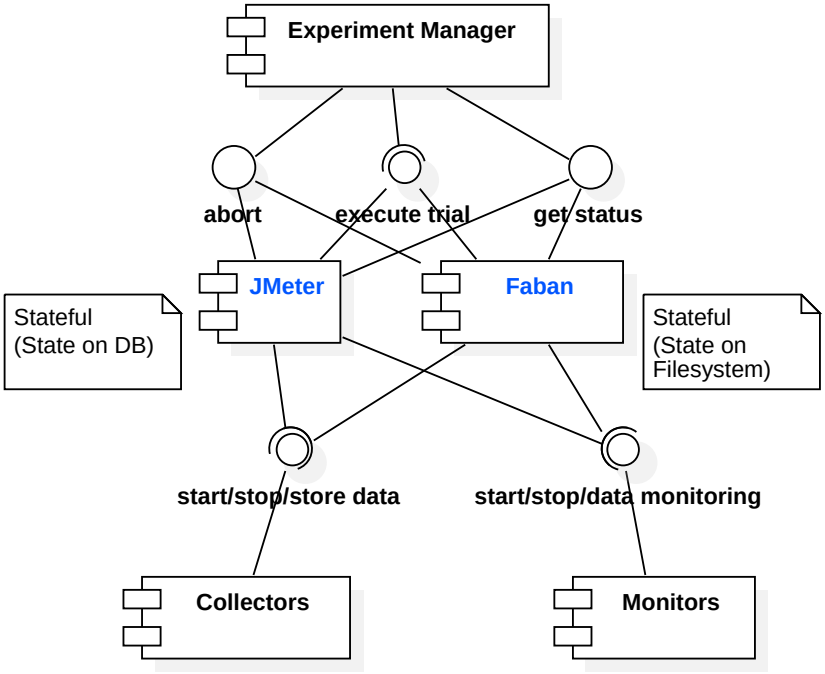

Figure 6.6. BenchFlow Framework: Trial Execution Frameworks Component Diagram

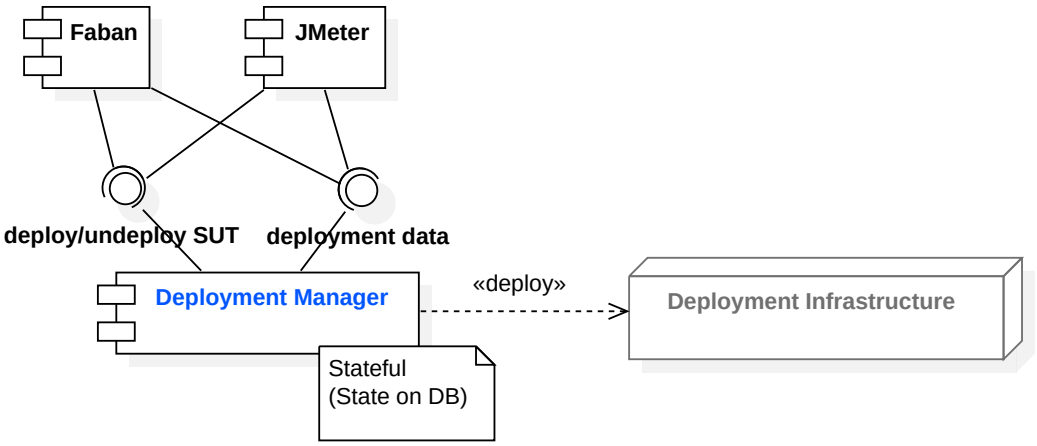

Figure 6.7. BenchFlow Framework: Deployment Manager Component Diagram

ecution to compensate them and improve the stability of the overall execution. The **Monitors** services are involved as part of the trial execution lifecycle, to ensure the SUT is ready to receive the load, and to assess the status of the SUT during and before deciding to collect performance data, using the **Collectors**. **Faban Drivers** and **JMeter Drivers** are imperative load drivers, realized according to the specification of the respective frameworks. At this level the declarative test specification provided using the DSL is not accessed, and all the data needed for the experiment's trials execution is codified as part of the drivers.

Deployment Manager - In Fig. 6.7 we present the deployment manager service, responsible for handling the deployment and undeployment of the SUT, monitors and collectors. The deployment manager is a core service and it is activated by the Faban/JMeter drivers, according to the trial execution lifecycle, and ensures all the mentioned services are correctly deployed. The deployment manager offers also interfaces to access deployment data generated dynamically at deployment time. The deployment data is needed by the Faban/JMeter drivers to know for example the endpoints of the monitors and collectors to call, as well as the endpoints of the SUT if those are generated dynamically.

Analysis Phase **Analysis Manager** - In Fig. 6.8 we present the analysis manager. The analysis manager is a core service and is responsible for scheduling performance data transformation (handled by **Data Transformers** re-

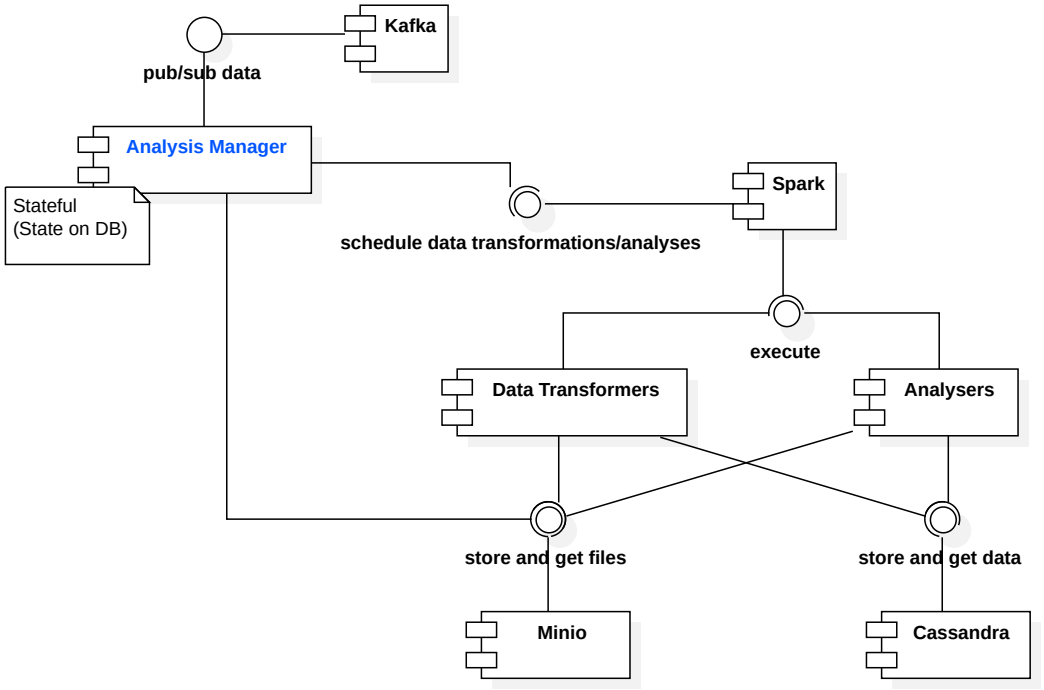

Figure 6.8. BenchFlow Framework: Analysis Manager Component Diagram

trieving performance data on *Minio* and storing transformed data on *Cassandra*), metrics and statistics computation (handled by **Analysers**, processing data stored on *Cassandra*) and managing the interdependencies among the same. The analysis manager is activated by messages published on *Kafka* by collectors, data transformers, and analyzers. We define metrics on different entities, as overviewed in Chap. 5, and we compute metrics and statistics after each trial, then for each experiment taking into account the results of all the executed trials and then at the test level considering the results of the different experiments. There are many interdependencies among the metrics computed on the same entity, as well as among metrics computed on different entities. More details are discussed in Sect. 6.3.5. The analysis manager takes care of scheduling the data transformers and analyzers for execution on *Spark*, according to the declared dependencies. The analysis manager accesses the declarative test specification by retrieving the test and experiments DSL from *Minio*, to identify the metrics and statistics to be computed and the entities on which to execute the computation.

The BenchFlow services across all the phases rely on the following third-party services:

Minio - storing unstructured data submitted by the users, generated by the BenchFlow services and collected during and after performance test execution. To facilitate the interaction with Minio we developed a library exposing to BenchFlow services, APIs to access data by providing generated entities identifiers. **Kafka** - exposing processing queues in a publish/subscribe manner, where different services write and read data to synchronize the computation on performance metrics. To facilitate the interaction with Kafka, we defined a dedicated software library. **Spark** - the infrastructure framework we rely on for data processing and metrics computation. Data transformers and analyzers are executed on top of Spark. To abstract common scheduling operations, we defined a dedicated software library. **Cassandra** - the distributed DBMS storing structured performance data, performance metrics, performance KPIs and statistics. To abstract common read/write operations, we defined a dedicated software library.

6.2.3 Sequence Diagrams

In this section we present the main interaction flows among the BenchFlow services and third-party services they depend on. The interaction flows we present are mostly related to test scheduling and test execution.
In Fig. 6.9 we present an overview of the main process the BenchFlow framework follows when a new test is submitted for execution. A *client*, being a user or a system utilizing the CLI or the APIs, submits a test bundle for execution to the test manager (operation #1) and receives back a *TestID* (#3). The *TestID* can be utilized for successive calls to the *test manager* APIs to retrieve data about the test execution. After receiving the test bundle, the test manager stores it on *Minio* for further use (operation #2). After storing the bundle, the test manager generates all the experiments to be executed and loops among the experiments interacting with the experiment manager after storing the generated experiment bundle (operation #4). For each experiment, the test manager delegates the *experiment manager* for execution, by identifying the experiment relying on the *ExperimentID* defined during the experiment generation (operation #5). The *experiment manager* retrieves the experiment definition from Minio (operation #6) and requests the *drivers maker* to generate the driver for all the trials of the experiment (operation #8). When the number of trials is determined dynamically, the *max_number_of_trials* value specified in the DSL specification is used. The *drivers maker* stores the generated drivers and configuration on Minio (operations #9-#12). After all the load drivers are generated and stored on Minio, the experiment manager

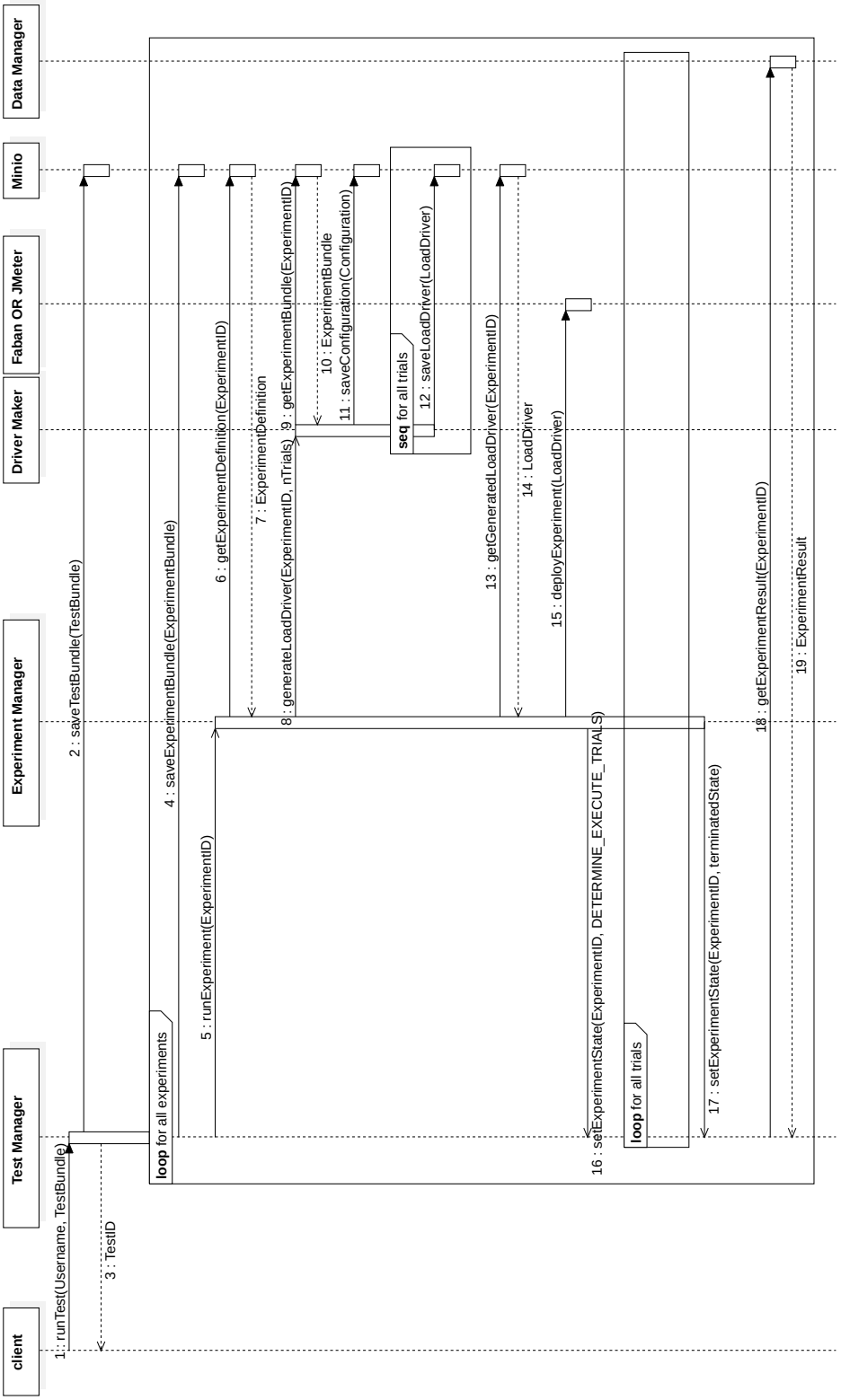

Figure 6.9. BenchFlow Framework: Main Processes Sequence Diagram

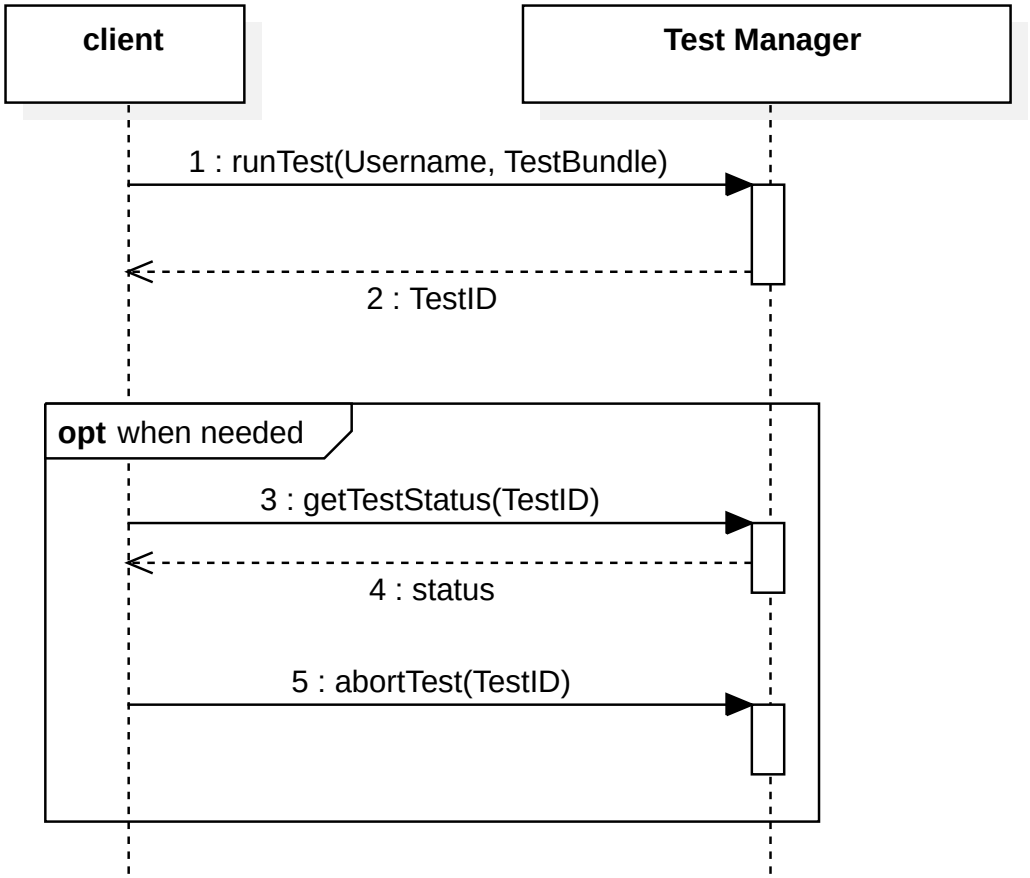

Figure 6.10. BenchFlow Framework: Client Process Sequence Diagram

submits to *Faban* or *JMeter* the load driver definition (operation #15), after retrieving the generated load driver from Minio (operation #13). Once the load driver definition is stored on Faban or JMeter, that might apply validation on the same, the experiment manager takes care of scheduling all the trials and signaling to the test manager about the state of the execution for the experiment (operations #16-#17). More details on this are provided in the following sequence diagrams. During the execution, the test manager requests the *data manager* to provide for experiments' results (operations #18-#19).
In Fig. 6.10 we present an overview of the possible interactions a *client* can perform with the *test manager*. The first interaction is usually about scheduling a new test bundle for execution (operation #1) and obtaining a test identifier (operation #2). Optionally, when needed, the client might also request to get the status of execution (operation #3) represented by a rich response containing

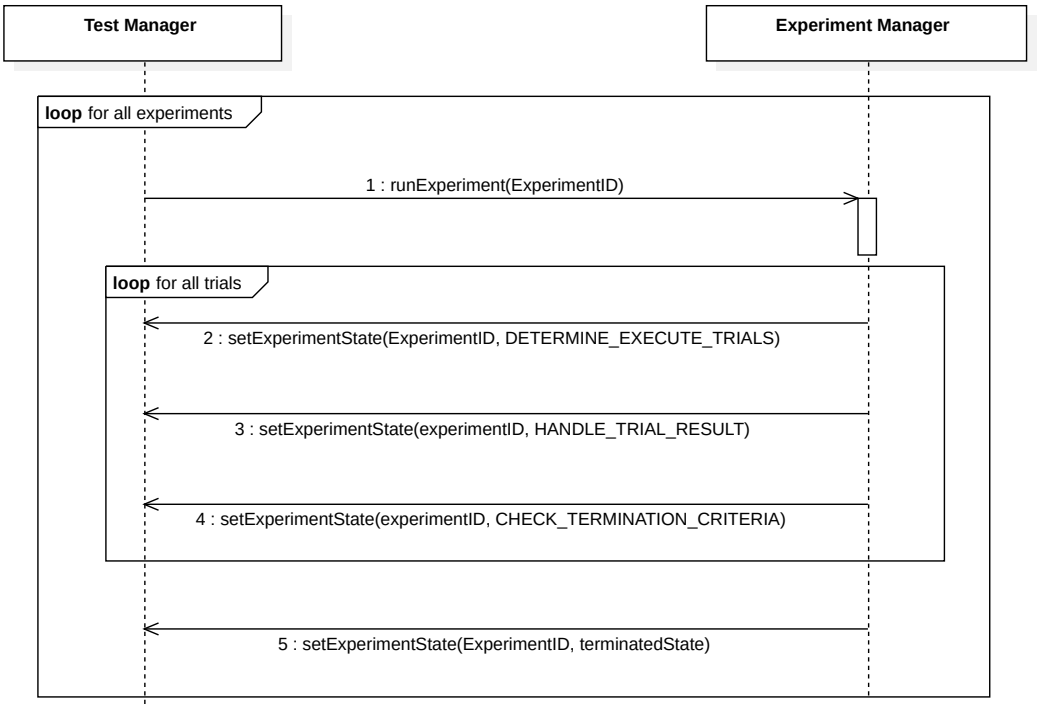

Figure 6.11. BenchFlow Framework: Test Manager Sequence Diagram

all the useful data (operation #4) and abort the test execution (operation #5). All the interactions happen to rely on the test identifier, generated taking into account the provided *Username* and ensuring it is unique.

Figure 6.11 presents a more detailed view on the main interactions the *test manager* performs with the *experiment manager* to request for experiments execution and getting updated about the state of execution of the experiment. As evident from Fig. 6.11, the test managers loops for all the experiments to be executed and, after submitting an experiment for execution (operation #1), it receives back from the *experiment manager* data about the experiment execution state. The actual state's identifiers depend on the internal behavior of the BenchFlow services and are discussed more in-depth in Sect. 6.2.4.

In Fig. 6.12 we highlight the main interactions between the *experiment manager* and the trial execution frameworks we support, namely *Faban* and *JMeter*. For all the trials, the experiment manager retrieves the generated load driver configuration from Minio (operation #1) and submits the trial and its configuration for execution to the target trial execution framework (operation #3). The target trial execution framework provides back a *RunID* (#4) that is stored by the experiment manager and mapped to the current *TrialID*. The experi-

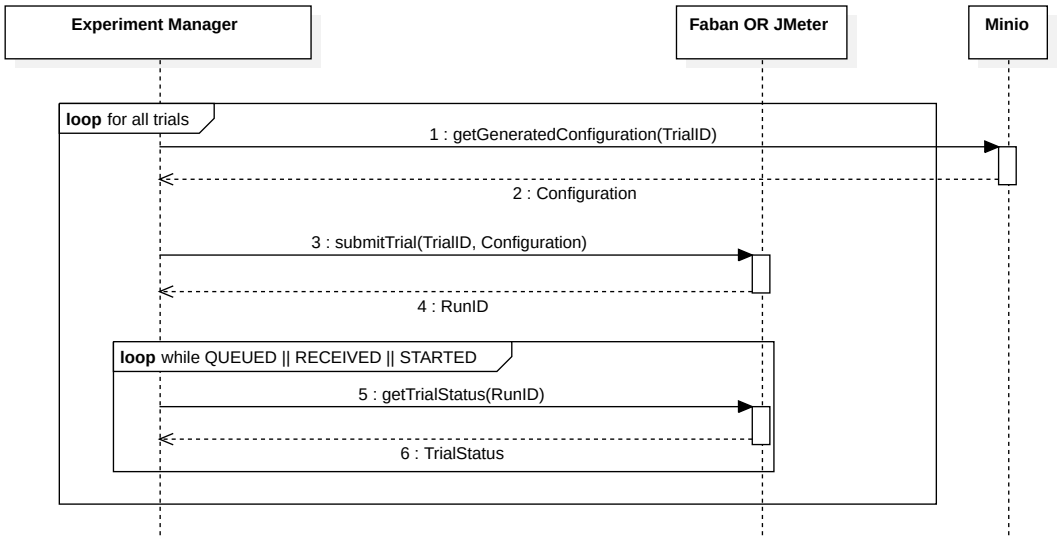

Figure 6.12. BenchFlow Framework: Experiment Manager Sequence Diagram

ment manager then starts a loop in which it polls the target trial execution framework (operation #5) for the status of the *RunID* until the trial execution is completed. The libraries we develop to interact with *JMeter* and *Faban* take care of uniforming the *TrialStatus* (#6). *Faban* and *JMeter* do not access data on *Minio*, but the experiment manager provides all the data they need. The reason is *Faban* and *JMeter* could be deployed on a different infrastructure than the other BenchFlow services, where *Minio* could not be reachable.
In Fig. 6.13 we highlight the interactions that either *Faban* or *JMeter* performs with BenchFlow services for automating the performance test execution. After accepting the load driver for execution, the trial execution framework interacts with the *deployment manager* to deploy the SUT, monitors, and collectors for the trial (operation #1). After receiving the request of deploying the SUT and other services for the trial, the *deployment manager* takes care of deploying them interacting with the underlying test infrastructure (operation #2). When the deployment has been completed, the deployment manager signals back the *Faban driver* or *JMeter* driver about the deployment status (operation #3). After receiving the deployment status from the *deployment manager*, the load driver pings (operation #4) the SUT on the default endpoints specified in the DSL and ensures it is correctly started and ready for receiving the load. After ensuring the SUT is ready to receive the load, the load driver starts all the monitor and collectors according to the current stage of the performance test execution and the moments in which the specific monitor and collector

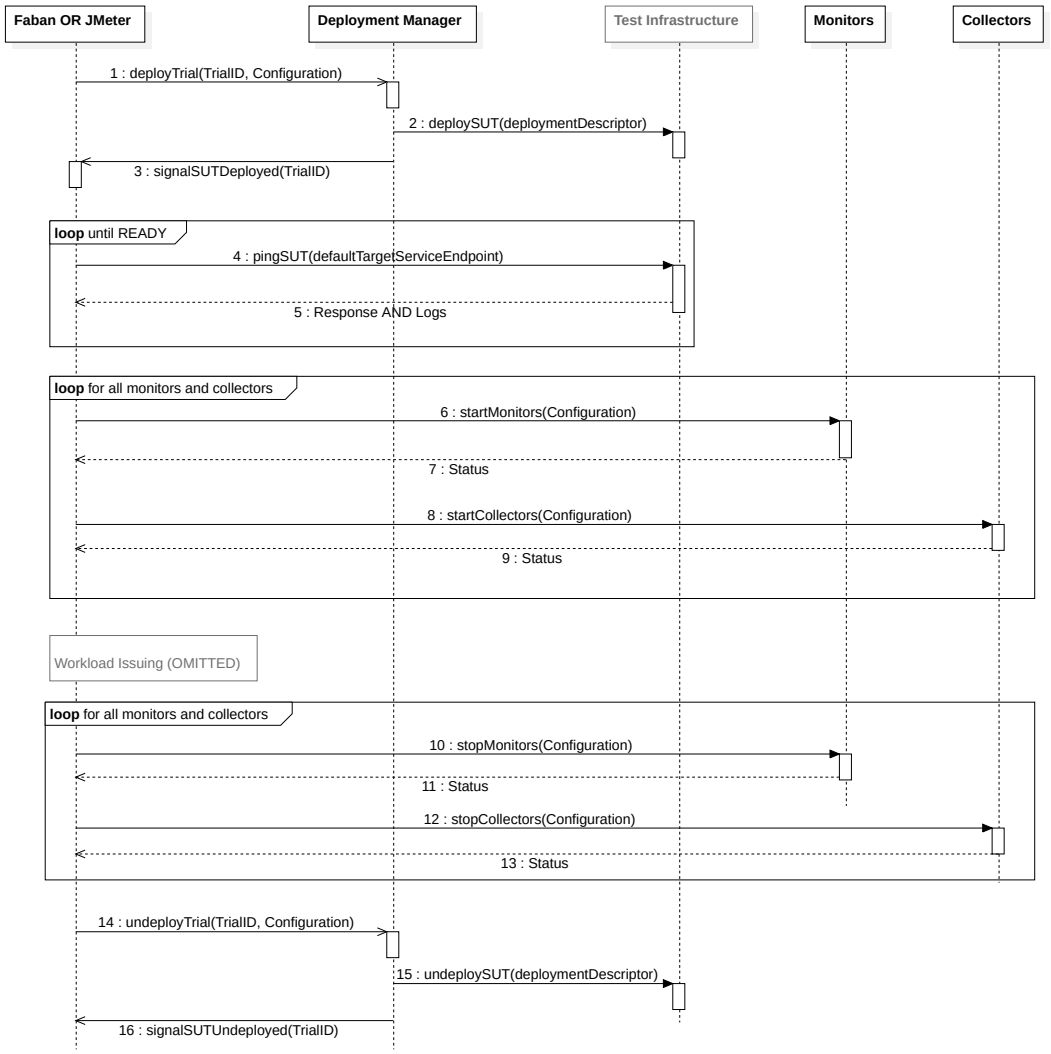

Figure 6.13. BenchFlow Framework: Trial Execution Frameworks Sequence Diagram

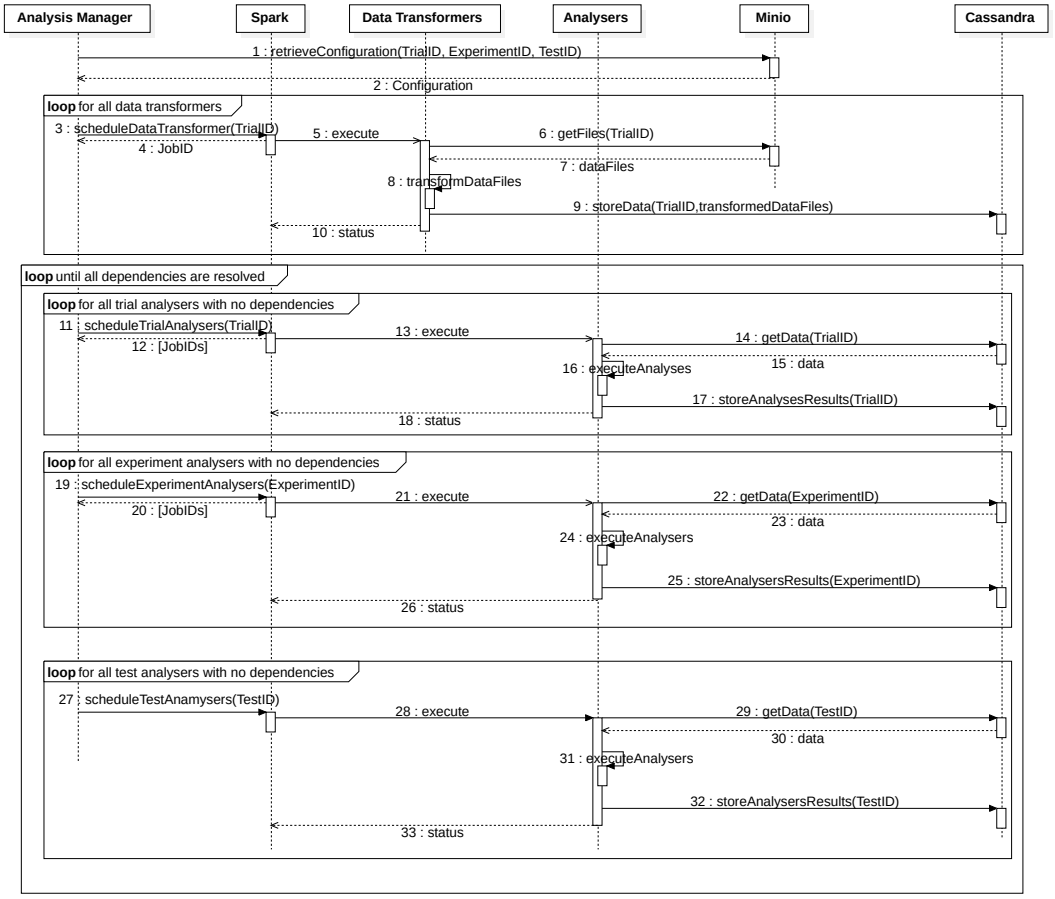

Figure 6.14. BenchFlow Framework: Analysis Manager Sequence Diagram

is expected to be started (operations #6-#9). The load drivers then start to issue the load to the SUT according to the specification. We omit this part in the diagram. At the end of the load driver execution, all the monitors and collectors are stopped (operations #10-#13), and the load driver takes care of requesting the *deployment manager* to undeploy the SUT and related services (operation #14). More details about monitors and collectors, and how they are started and stopped during the trial execution lifecycle, are provided in Sect. 6.2.4.

In Fig. 6.14 we present the main interactions between the *analysis manager* and its dependent services. The analysis manager sequence highlighted in the diagram represents the interactions the analysis manager performs to transform data and compute metrics and statistics for the trials, the experiments, and the tests. The sequence is activated each time new work is present in dedicated

working queues on *Kafka*. The *analysis manager* retrieves the DSL specification for the test, experiments, and trials of interest from *Minio* (operation #1). According to the retrieved specifications, it then schedules *data transformers* for the current trial on *Spark* (operation #3). *Spark* takes care of executing the *data transformers*, which retrieve performance data for the current trial to be processed from *Minio* (operation #6) and store transformed data on *Cassandra* (operation #9). The *analysis manager* schedules all the *data transformers* for the collected performance data and tracks the execution relying on *JobIDs* returned by *Spark*. After transforming all the data, the *analysis manager* starts a loop to ensure all the *analyzers* for the current trial (operations #11-#18), the experiment (operations #19-#26) to which the current trial belongs to and the test (operation #27-#33) to which the experiment belongs to are scheduled. The control loop ensures all the metrics are computed, and all the interdependencies among the metrics are resolved until all the metrics specified in the declarative specification are computed. The *analyzers* retrieve data from *Cassandra* and store computed metrics and statistics on *Cassandra*. More details about interdependencies among *data transformers* and *analyzers* for different entities, and how they are scheduled, are provided in Sect. 6.3.5.

6.2.4 Automation Life-cycles' State Machines Diagrams

In this section, we present the state machines diagrams, representing the core business logic of the main services realizing the BenchFlow framework. The state machines allow us to codify the state of the system during the performance test automation process, and can be configured in their behavior using the DSL part of the proposed approach and events generated during the automation process, such as successful/failing experiments execution and their state, as well as computed performance metrics and KPIs or failure in computing the same. We provide five main state machines, implementing the business logic of the following services:

a) the *test manager*;

b) the *experiment manager*;

c) the *load drivers* executed by the *trial execution frameworks*;

d) the *deployment manager*;

e) the *analysis manager*.

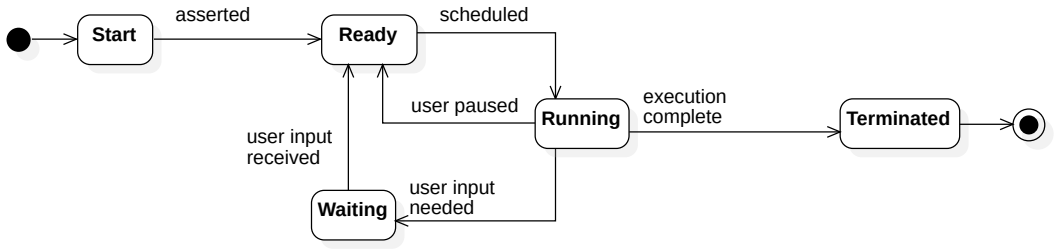

Figure 6.15. BenchFlow Framework: State Machine Overview

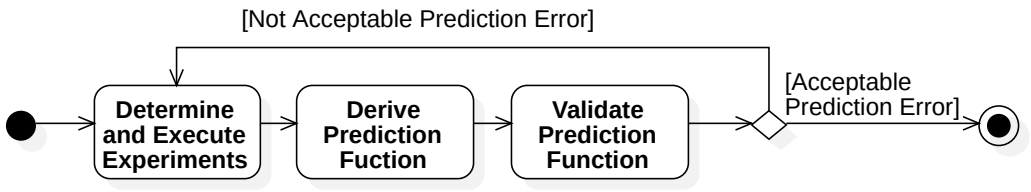

Figure 6.16. BenchFlow Framework: Prediction Life-cycle State Machine Overview

The high-level states presented in Fig. 6.15 are inspired by the scheduling of processes in an operating system [Silberschatz et al., 2013] and are: `start` (named "new" in the referenced diagram), `ready`, `running`, `waiting` and `terminated`. All the implemented state diagrams refer to the mentioned states and defined sub-state behaviors according to the business logic of the service.
Part of the DSL allows also the specification of exploration strategies based on prediction models. In Fig. 6.16 we present an overview of a standard prediction lifecycle, presenting the states and the transition among the states.
The prediction lifecycle is integrated as part of the running state of the test manager state machine, presented in Fig. 6.17.
In Fig. 6.17 we present the state machine implementing the *Goal Exploration* lifecycle driven by the *test manager*. This lifecycle is instantiated each time a new test is submitted for execution, and multiple lifecycle instances can be executing at the same time. The `Start` state is reached after the test or test suite bundle has been verified as syntactically and semantically correct, and automatically generated sections are added to the specification, e.g., data collectors. Syntax and semantics correctness is verified relying on the library introduced in Sect. 5.6.4, taking also care of generating the exploration space and the SUT and related services deployment descriptors (e.g., for monitors and collectors) for the test execution. When errors occur during the initialization phase or warnings have to be reported to the user, the test is not moved

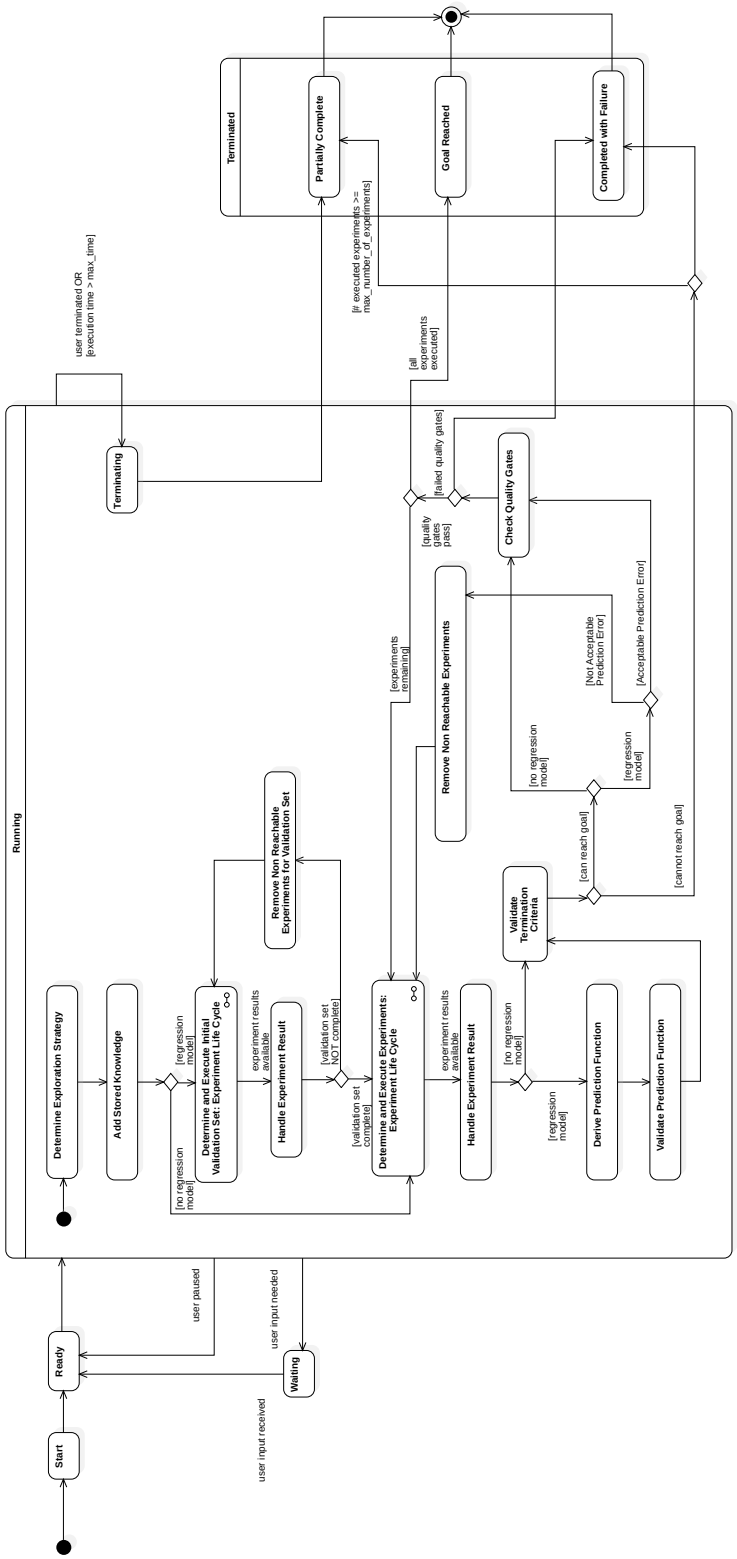

Figure 6.17. BenchFlow Framework: Test Manager State Machine

to the `Start` state. The `Start` state setups the BenchFlow framework to be ready for test execution (i.e., stores relevant data to be accessible in the next states, e.g. part of the test specification often accessed across the test lifecycle). If no errors happen in the `Start` state, the test is moved to the `Ready` state and is available to be scheduled for execution by the *test manager*. If errors are encountered during the syntactic or semantics validation or in the `Start` state, the stored data are deleted, and the user is warned the test can not be scheduled for execution. When there are resources available for execution, the test is moved from the `Ready` state to the `Running` state, where the actual *Goal Exploration* happens. In the running state, the *test manager* generates the experiment bundles, relying on the library introduced in Sect. 5.6.4. The user, or a system, could decide to pause the test. In this case, the test is moved to the `Ready` state if no user input is required, or to the `Waiting` state if the *test manager* has to wait for user input before proceeding. Once the execution of the test is completed, it is moved to the `Terminated` state, representing a final test state after which the state of the test can change only if re-started by the user, e.g. because the user decides to manually restart a `Completed with Failure` test. The running state is realized by a sub-state machine, with the following states: 1) `Determine Exploration Strategy`; 2) `Add Stored Knowledge`; 3) `Determine and Execute Initial Validation Set`; 4) `Handle Experiment Result`; 5) `Remove Non Reachable Experiments`; 6) `Determine and Execute Experiments`; 7) `Handle Experiment Result`; 8) `Derive Prediction Function`; 9) `Validate Prediction Function`; 10) `Validate Termination Criteria`; 11) `Check Quality Gates`; 12) `Remove Non Reachable Experiments`; 13) `Terminating`. Most of the states of the state machine are mapped to the DSL entities and represent the execution semantics driven by the declarative specification described in Sect. 5.4.

The `Determine Exploration Strategy` state is used to read the *Exploration Strategy* for experiment selection specified by the user in the test specification. The `Add Stored Knowledge` is used for retrieving the knowledge about the performance of the SUT from the execution of previous tests. We currently support reuse only for performance metrics computed out of tests and experiments defining the same specification. After a user has been executing tests for a given SUT for some time, it is likely that some of the experiments will be overlapping and thus the knowledge stored in the database about experiments result can be reused to avoid executing a given experiment.

The `Determine and Execute Initial Validation Set` is activated when the *Exploration Strategy* involves a prediction model. This state takes care of selecting an initial set of experiments in the exploration space, to be executed

and included as part of the underlying model validation set. The only validation strategy we support is currently one selecting experiments to be included in the validation set at random. The actual execution of the experiments is handled by the experiment manager and presented later in this section.
The `Handle Experiment Result` state is activated each time new experiments are complete, and takes into account the final execution state of the experiment, as presented in Fig. 6.20. The experiment manager updates the test manager about the state of the experiment and its trials, and when the experiment reaches the final state the test manager checks for experiment results to be taken into account for deciding how to continue with the *Goal Exploration*.
The `Handle Experiment Result` state ensures that the result is retrieved and saved for easy access in other parts of the lifecycle, e.g., for use in the `Check Quality Gates` state. When the validation set is complete, the state machine is moved to the `Determine and Execute Experiments` state. If experiment results are not received within a given maximum time, e.g., due to failure in the execution or errors in data analysis, then, according to the context in which the `Handle Experiment Result` is reached, a different behavior occurs. When the `Handle Experiment Result` state is executed after the `Determine and Execute Initial Validation Set` state, in case experiment results are not retrieved, the `Remove Non Reachable Experiments for Validation Set` state is activated and the experiment is removed from the validation set. When the `Handle Experiment Result` state is reached after the `Determine and Execute Experiments` state, the actual behavior depends on the following states and the current experiment is marked as missing results.
The `Determine and Execute Experiments` state selects the experiment to be executed according to the test goal, the exploration space, and the selection strategy. The actual experiment execution is handled by the experiment manager, and each time a new experiment's results are available the `Handle Experiment Result` state is activated.
When a predictive exploration strategy is involved, the `Derive Prediction Function` state first and the `Validate Prediction Function` state next are activated. The first one updates the prediction function of the underlying prediction model integrating the new results of the experiment. The second one, validates the updated prediction function with the validation set, to compute the prediction and prediction error metrics.
The `Validate Termination Criteria` state is activated after a new experiment's results are received, or after updating the model prediction function in case of predictive exploration. This state validates the termination criteria according to the specification in the submitted test bundle for the executing

test, and also accounts for possible fatal failures during the experiment execution making it impossible to reach the stated goal (e.g., the SUT can not be deployed). The termination criteria determine whether or not the test execution can proceed. If the termination criteria verification determines the test has to be stopped before the declared number of experiments is executed, then the final *Terminated* inner state is set as the `Completed with Failure` state. If the *max_number_of_experiments* termination criterion is triggered and no other criteria are triggered, then the final *Terminated* inner state is set as the `Partially Complete` state, because no actual failures occurred. When experiment results are not retrieved in the `Handle Experiment Result` state, the experiment is considered in the percentage computed for evaluating the *max_failed_experiments* termination criterion.
If the termination criteria do not stop the execution, then the quality gates are evaluated by activating the `Check Quality Gates` state, if no prediction model is involved or the prediction model has an acceptable prediction error. The quality gates are evaluated on metrics computed on different entities, and must always pass for the test to continue. This allows us to mark the test as `Completed with Failure` as soon as the quality gates part of the specification are not met. The evaluation of the quality gates depends on the actual metrics part of the gates and the availability of the metric. If the metric is not available because the `Handle Experiment Result` state has not been able to retrieve it, the quality gate evaluation is skipped because they can not be evaluated. In the latter case, the decision about terminating or continuing the execution is delegated to the termination criteria, ensuring experiments do not keep failing over a certain threshold (*max_failed_experiments*) or the test does not execute for more than a specified amount of time (*max_time*)
If the quality gates are met, and there are still experiments to be executed, the lifecycle moves back to the `Determine and Execute Experiments` state. If quality gates are met, and all the experiments have been executed, the test is marked as `Goal Reached` meaning the test successfully reached the goal declared as part of the specification.
When a prediction model is involved, before evaluating the quality gates, we ensure the prediction model has an acceptable prediction error, according to the *mean_absolute_error* quality gate. This allows the evaluation of quality gates only when the represented performance has an acceptable precision, even if potentially more experiments are executed before evaluating all the quality gates. In parallel with the scheduling of experiments part of the test, we ensure to `Remove Non Reachable Experiments` according to the prediction model validation results, to speed up the exploration of the performance space. Examples

of non-reachable experiments are, e.g., in dimensions of the performance space where experiments are prone to fail. For instance, if an experiment has failed because of limited memory available, and there are other experiments in the exploration space with less memory configured, then we heuristically assume that these experiments also would fail and therefore there is no need for them to be executed.
Some termination criteria the user can specify relying on the DSL are based on time. In this case, the BenchFlow framework ensures when the time threshold is reached, the test is moved to the `Terminating` state. To guarantee that almost terminating tests have the opportunity to do so, we account for a buffer in the evaluation of the actions to take in the `Terminating` state. BenchFlow, before scheduling an experiment, always checks for the remaining time so that experiments are not scheduled if they have a high chance of getting aborted because of time constraints. When the `Terminating` state is activated, we check for running experiments and remaining time for their execution, and we wait for their completion, without scheduling new ones, if the remaining time exceeds at most 30 minutes or 5% of the stated maximum execution time. In this case, the experiment is allowed to complete the execution, and termination criteria as well as quality gates are evaluated. When that is not the case, the `Terminating` state triggers an abortion of the experiment, consequently terminating all its trials, and the SUT and relates services undeployment, and moves the test to the `Partially Complete` state. Metrics computation handled by the analysis manager are completely executed, so that data is transformed and metrics are computed for all the completed trials, in case the user decides to resume the test. The same state is also reached when the user decides to abort the test execution. The `Terminated` state represents the final state of the test execution. As presented in this section, the final sub-state can be one of `Partially Complete`, `Goal Reached`, or `Completed with Failure`. The final state is set based on the evaluations on the test results made by sub-states of the `Running` state. The final state always reports also details the state of the test execution, so that it can be reported to the user.
We differentiate between three main types of tests, all executed relying on the lifecycle presented in Fig. 6.17: *single experiment*, *exploration without regression model*, and *exploration with a regression model*. The *exploration with regression model* type is represented by the entire lifecycle reported in Fig. 6.17. An example of a test following the entire lifecycle is the one reported in Listing 5.15. Depending on the type or the result after executing the business logic belonging to a given state, the test takes different paths in the lifecycle.
In Fig. 6.18 we present the states activated when a *single experiment* test type

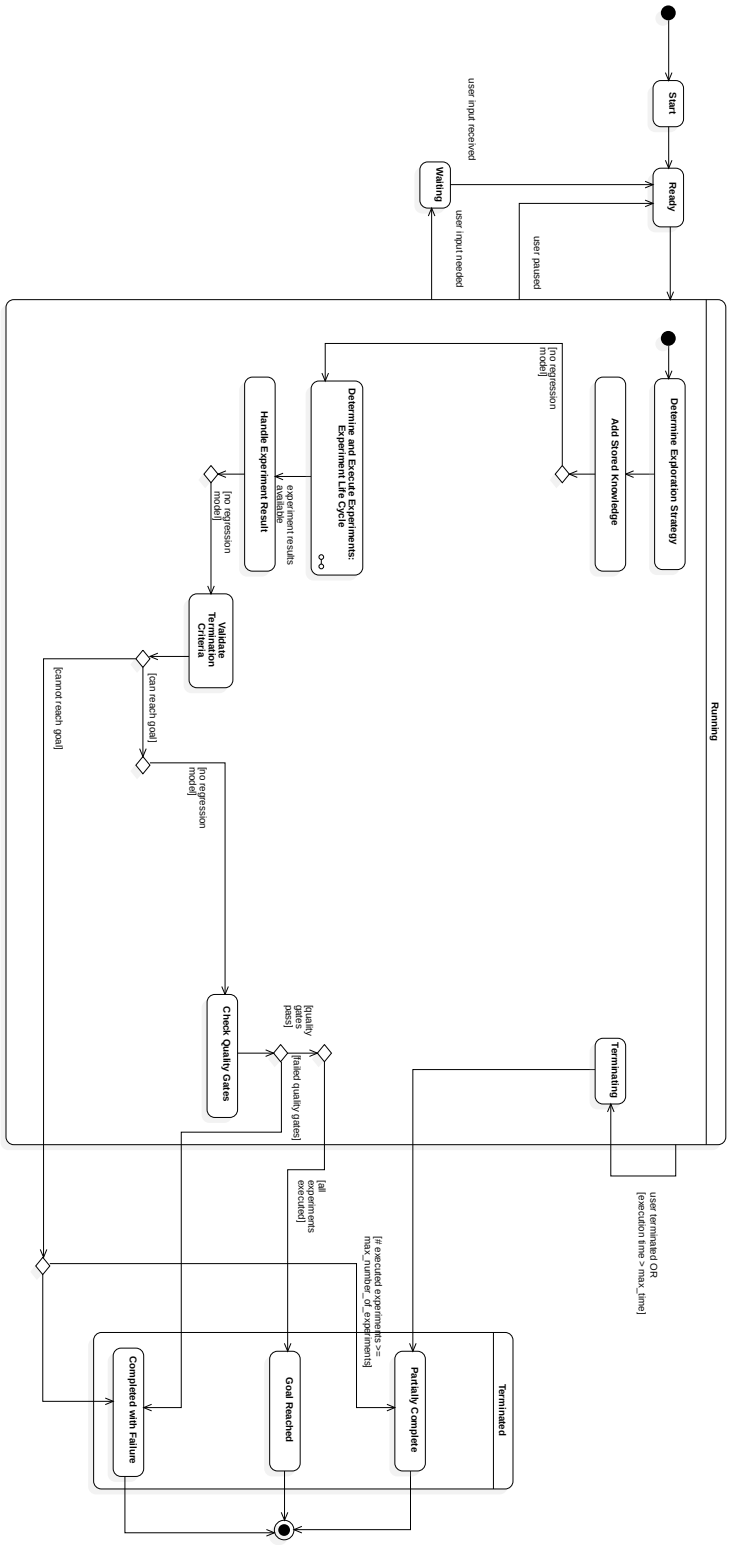

Figure 6.18. BenchFlow Framework: Single Experiment State Machine

is executed. An example of a test following this lifecycle is a load test, as the one presented in Listing 5.27. This is the simplest test type that can be executed, as evident from the limited amount of states getting activated. When a test with a single experiment is executed, after the experiment execution, the results are evaluated, termination criteria are verified and the final terminated state is decided according to the quality gates evaluation.
In Fig. 6.19 we present the states activated when an *exploration without regression model* test type is executed. An example of a test following this lifecycle is a configuration test, like the one presented in Listing 5.31. The activated states are the same as per the test presented in Fig. 6.18. In this case, since more experiments have to be executed, after the quality gates evaluation the lifecycle loops back to the `Determine and Execute Experiments` state.
In Fig. 6.20 we present the state machine implementing the experiment execution lifecycle driven by the *experiment manager*. The state machine is activated for a given test when the test lifecycle is in one of the following two states: `Determine and Execute Initial Validation Set`, or `Determine and Execute Experiments`. The experiment lifecycle handles the execution of trials, and re-execution of the same in case of failures, and interacts with the test manager to report back with the current execution status of the experiment as presented in 6.11. As per the test lifecycle, also for the experiment lifecycle, the `Running` state is the most complex one in terms of sub-states. Differently from the test lifecycle, for the experiment lifecycle, we omit the `Waiting`, because we assume that an experiment should be completed once it is scheduled for execution. In the `Start` state the experiment is prepared for execution and frequently accessed data is stored in the database. If there are any errors in the preparation the experiment is directly `Terminated` and moved in the `Error` state since it can not be executed.
When the experiment is ready, it is moved to the `Ready` state, waiting to be scheduled for `Running`. The running state is realized by a sub-state machine, with the following states: 1) `Determine and Execute Trials`; 2) `Handle Trial Result`, with sub-states `Handle Trial Result` and `ReExecute Trial`; 3) `Validate Termination Criteria`; 4) `Terminating`.
In the `Determine and Execute Trials` state, the next trial to be executed is determined and scheduled for execution on the trial execution framework. When the execution results are available the experiment moves to the `Handle Trial Result` state. The `Handle Trial Result` state checks for availability of trial results, or for incurred errors, similar to what the `Handle Experiment Result` state does in the test lifecycle. If the trial completes the execution with success, and results are available, then the experiment is moved to the `Validate`

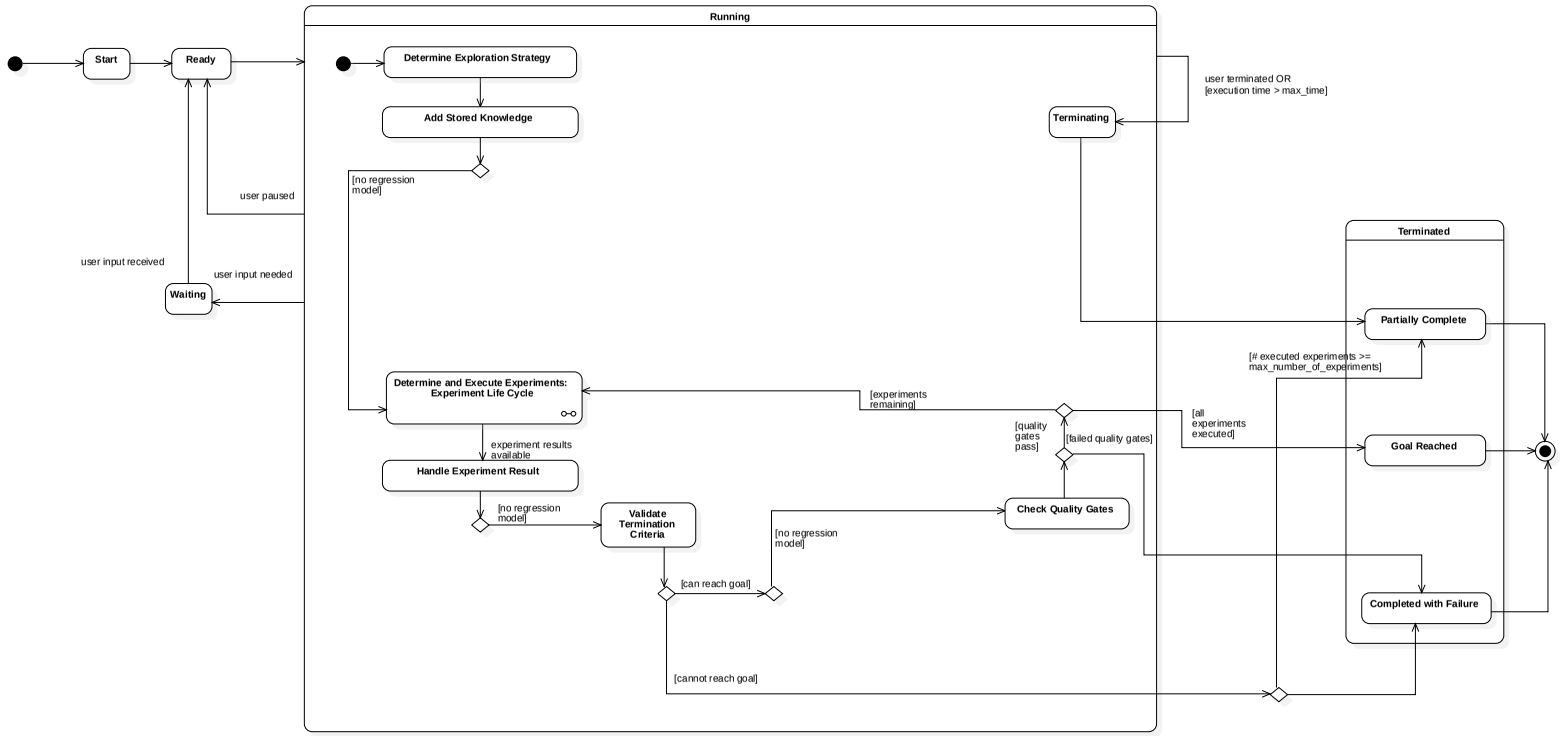

Figure 6.19. BenchFlow Framework: Exhaustive Exploration with No Regression Model State Machine

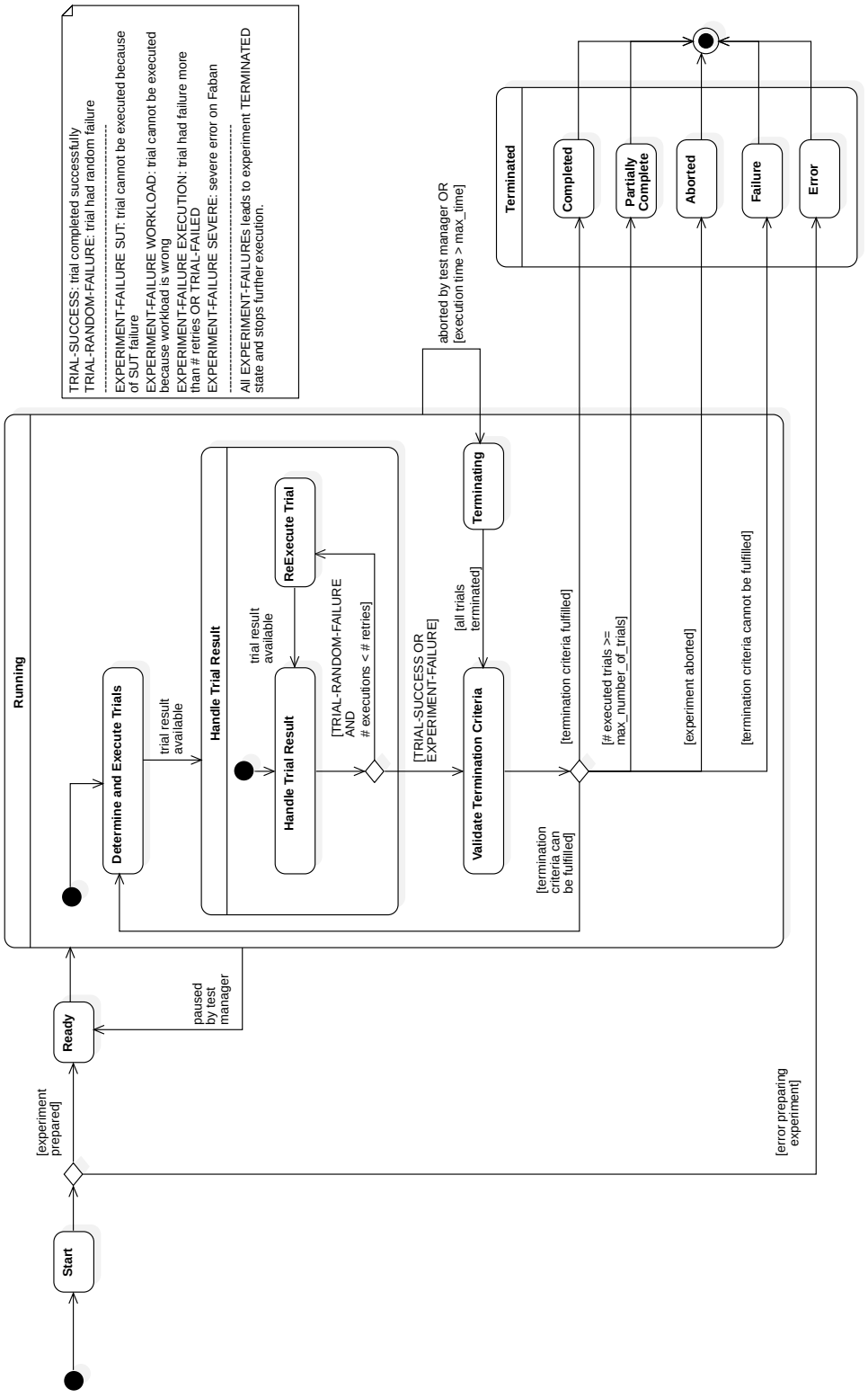

Figure 6.20. BenchFlow Framework: Experiment Manager State Machine

`Termination Criteria` state. The `Validate Termination Criteria` state validate the termination criteria applied to the experiment and its trials. For the experiments, according to the experiment meta-model presented in Sect. 5.3.2, the termination criteria control the maximum number of time the experiment can execute, the maximum number of total or failed trials, as well as allows to dynamically determine the number of trials according to the precision of results obtained for metrics defined on workloads and services. If more trials remain to be scheduled, the lifecycle goes back to the `Determine and Execute Trials` state. The user, or a system, could decide to pause the test, and consequently all the experiments related to the test. In such a case, the experiment is moved to the `Ready` state. The final `Terminated` sub-state the experiment reaches, depends on the termination criteria validation. In the following we discuss the conditions determining the actual final state for the experiment:

a) `Completed`, reached when all the trials are successfully executed. If termination criteria dynamically determining the number of trials are defined, the state is reached if the expected confidence interval for the defined metrics is guaranteed;

b) `Partially Complete`, reached when the execution is suspended because the total number of executed trials exceeds the maximum number of allowed trials;

c) `Aborted`, reached when the experiment is aborted by the test manager;

d) `Failure`, reached when termination criteria can not be fulfilled, e.g., when the number of failing trials exceeds the maximum percentage of failed trials;

e) `Error`, reached if the experiment preparation incurs in errors.

When the test manager requests the experiment to abort the experiment execution, or when the maximum time the experiment is allowed to execute is exceeded, the `Terminating` state is activated. As per the test, also for the experiment, some time buffer is given for currently executing trial to complete, according to the remaining execution time of such trial. The `Terminating` state terminates all the running trials, and then moves the lifecycle to the `Validate Termination Criteria` state to determine the actual final state of the experiment execution.
In case of an error in the trial execution, different actions are taken, according to the incurred error, and they impact the state transitions occurring in

the experiment lifecycle. We define all the possible types of failure the test, experiment, and trial execution could experience, as reported in the note in Fig. 6.20. Prevention of possible failures and handling of unexpected failures is very important in highly automated processes, such as the ones implemented in BenchFlow.
In the case, the trial fails and there is no data about the reason for the failure, we consider it a *TRIAL-RANDOM-FAILURE*, and we try to re-execute the trial for a predefined number of times set to three by default. If the trial keeps on failing after being re-executed we determine an *EXPERIMENT-FAILURE*, because it can be assumed that also other trials of the same experiment will fail, considering they all have the same configuration. In the `Validate Termination Criteria` state the termination criteria are validated. If we incur in an *EXPERIMENT-FAILURE* we consider the experiment to have failed and is then terminated in the `Failure` state, because no other trials can be scheduled. Besides the trial leading to an *EXPERIMENT-FAILURE* we also consider other situations leading to such a failure, so that the experiment execution is stopped as soon as possible in case it is not going to be successfully executed. An *EXPERIMENT-FAILURE* can incur also when:

a) the SUT or related services experience a failure at deployment time;

b) the SUT or related services experience a failure at runtime;

c) the SUT is not ready for receiving the load after a certain amount of time (i.e., the *sut_ready_log_check* fails);

d) the workload specification is not correct, e.g., because the specified SUT endpoint is wrongly specified;

e) the trial execution framework has an unexpected severe error preventing the execution of trials, e.g. because load drivers can not be scheduled or executed.

As performance tests often require a significant amount of execution time, automatic failure handling is crucial in CSDL, and failures should therefore be handled as soon as possible. In the case of *EXPERIMENT-FAILURE* #a, #c the test execution is suspended as well, because the goal is considered not possible to be reached.
In Fig. 6.21 we present the state machine implementing the experiment trial execution lifecycle driven by the *load drivers* deployed on the trial execution framework (either *Faban* or *JMeter*). As for the other lifecycles presented in

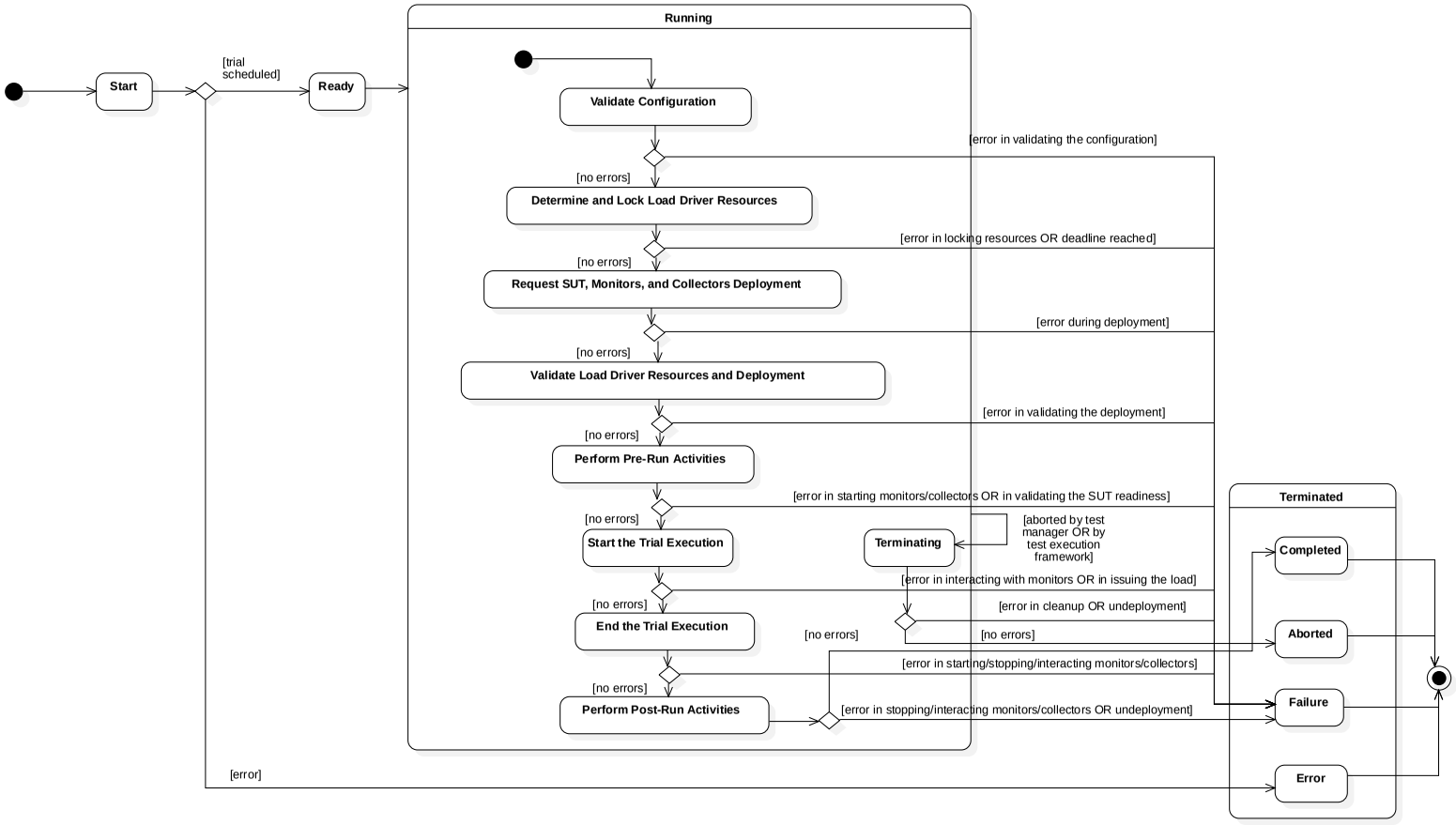

Figure 6.21. BenchFlow Framework: Trial Execution Frameworks State Machine

this section, the `Running` state is the most complex one. The experiment trial execution lifecycle performs all the automation activities to be executed before, during, and after the issuing of the load specified in the test specification. Among the different activities, this lifecycle is responsible for correctly starting and stopping *monitors* and *collectors* specified in the *data_collection* section of the test specification, according to the moment in the test lifecycle the different monitors and collectors are expected to start. *Monitors* and *collectors* are of different types, and according to the type, they must (or can) be started in specific moments of the trial execution lifecycle to serve their purposes. The *monitors* can be queried in the following moments of the lifecycle (which we name *phase*s):

a) *start*, for monitors involved before the issuing of the load to the SUT;

b) *end*, for monitors involved after the issuing of the load to the SUT;

c) *all*, for monitors involved before or after the issuing of the load to the SUT.

The *collectors* can be queried in the following moments of the lifecycle:

a) *end*, for collectors involved after the issuing of the load to the SUT;

b) *all*, for collectors involved before or after the issuing of the load to the SUT.

In Sect. 6.2.5 we provide an in-depth overview of all the available *monitors* and *collectors* services part of BenchFlow and more details on the different phases. The experiment trial execution lifecycle involves the following states:

1) `Start`, the load driver is validated by the trial execution framework, and in case of error, the trial execution state is moved to the `Error` termination state;

2) `Ready`, reached when the trial is ready for execution;

3) `Running`, activated when the trial is scheduled for execution by the trial execution framework;

4) `Terminated`, reached at the end of the trial execution.

The `Running` state is realized by a sub-state machine, with the following states: 1) `Validate Configuration`, validating the load driver runtime configuration

after the load driver has been deployed on the trial execution framework by the framework itself; 2) `Determine and Lock Load Driver Resources`, evaluating the load driver configuration for the number of simulated users requests, and ensure the load infrastructure has sufficient resource. A time-bound is set to wait for resource availability, although the trial is scheduled by the trial execution framework (i.e., moved to the `Running` state) only when sufficient resources are available and reserved for the trial execution; 3) `Request SUT, Monitors and Collectors Deployment`, requesting the deployment manager to deploy the SUT as well as monitoring and collection services on the test execution infrastructure. All monitors and collectors services are deployed before the actual trial execution, even if they are involved in a later state of the lifecycle. This ensures that in case of deployment errors, the trial is stopped before being executed; 4) `Validate Load Driver Resources and Deployment`, validating the correct deployment of the load drivers, the SUT, monitors and collectors; 5) `Perform Pre-Run Activities`, starting monitors and collectors declaring the running phase as *all* and executing the *sut_ready_log_check*; 6) `Start the Trial Execution`, issuing the actual load to the SUT and before initiating the load issuing, it also starts the monitors declaring the running phase as *start*; 7) `End the Trial Execution`, starting monitors and collectors declaring the running phase as *end*, and then stopping them at the end of their execution. It also stops monitors declaring the running phase as *start*; 8) `Perform Post-Run Activities`, stopping monitors and collectors declaring the running phase as *all*, at the end of their execution. It also requests the deployment manager to undeploy the SUT and related services, because the trial execution can be considered complete; 9) `Terminating`, activated when the experiment lifecycle requests to abort the trial execution or because the load driver gets killed by the trial execution framework. The state execution ensures the deployment manager is requested to undeploy the SUT and related services, as well as cleans up the trial execution framework resources allocated to the running load driver. In this case, the final trial execution state is marked as `Aborted`. If no errors incur during the execution, the final `Completed` state is reached and trial execution framework resources are unlocked. When error occurs in the `Running` state, the `Failure` final state is reached. Errors can occur in the following states: 1) `Validate Configuration`, when validating the load driver runtime configuration errors might incur; 2) `Determine and Lock Load Driver Resources`, when trying to lock resources for the trial execution, resource unavailability might trigger a deadline resulting in an error in locking resources for execution; 3) `Request SUT, Monitors and Collectors Deployment`, when an error during the deployment of the SUT and related services and/or the load

driver incur; 4) `Validate Load Driver Resources and Deployment`, in case the SUT and its related services, as well the load driver deployments result invalid. One example is when the services are deployed but not available; 5) `Perform Pre-Run Activities`, when errors in starting/interacting with the monitors and collectors or when validating the SUT readiness to receive load incur; 6) `Start the Trial Execution`, when errors during the issuing of the load to the SUT or in starting/interacting with the monitors incur; 7) `End the Trial Execution`, when errors in starting/stopping/interacting with the monitors and collectors incur; 8) `Perform Post-Run Activities`, when errors in stopping/interacting with the monitors and collectors incur or during the SUT and related services undeployment; 9) `Terminating`, when errors during the cleanup of the trial execution framework or the SUT and related services undeployment. Given most of the errors can be temporary, e.g., because of network issues when interacting with the SUT, monitors, or collectors all the services implement retry policies to be resilient.
In Fig. 6.22 we present the state machine implementing the SUT and related services lifecycle driven by the *deployment manager*.
The SUT and related services deployment lifecycle involves the following states:

1) `Start`, the deployment descriptor is received from the load driver and validated and in case of error, the deployment execution state is moved to the `Error` termination state;

2) `Ready`, reached when the SUT and related services are ready for deployment;

3) `Running`, activated when the SUT and related service are deployed;

4) `Terminated`, reached at the end of the deployment lifecycle, after the SUT and related services undeployment.

The `Running` state is realized by a sub-state machine, with the following states: 1) `Determine and Lock SUT Resources`, ensuring the requested resource for SUT deployment according to optional requirement about the server where to deploy the services are available; 2) `Validate Resource and Deploy SUT, Monitors and Collectors`, ensures the locked resources are ready for SUT and related services deployment and proceeds with the deployment ensuring the monitors and collectors are healthy after the deployment; 3) `Ensure the SUT is Healthy`, makes sure the SUT services are healthy by querying the deployment infrastructure and ensuring the state of all the services is healthy; 4) `Undeploy SUT, Monitors and Collectors and Release Resources`

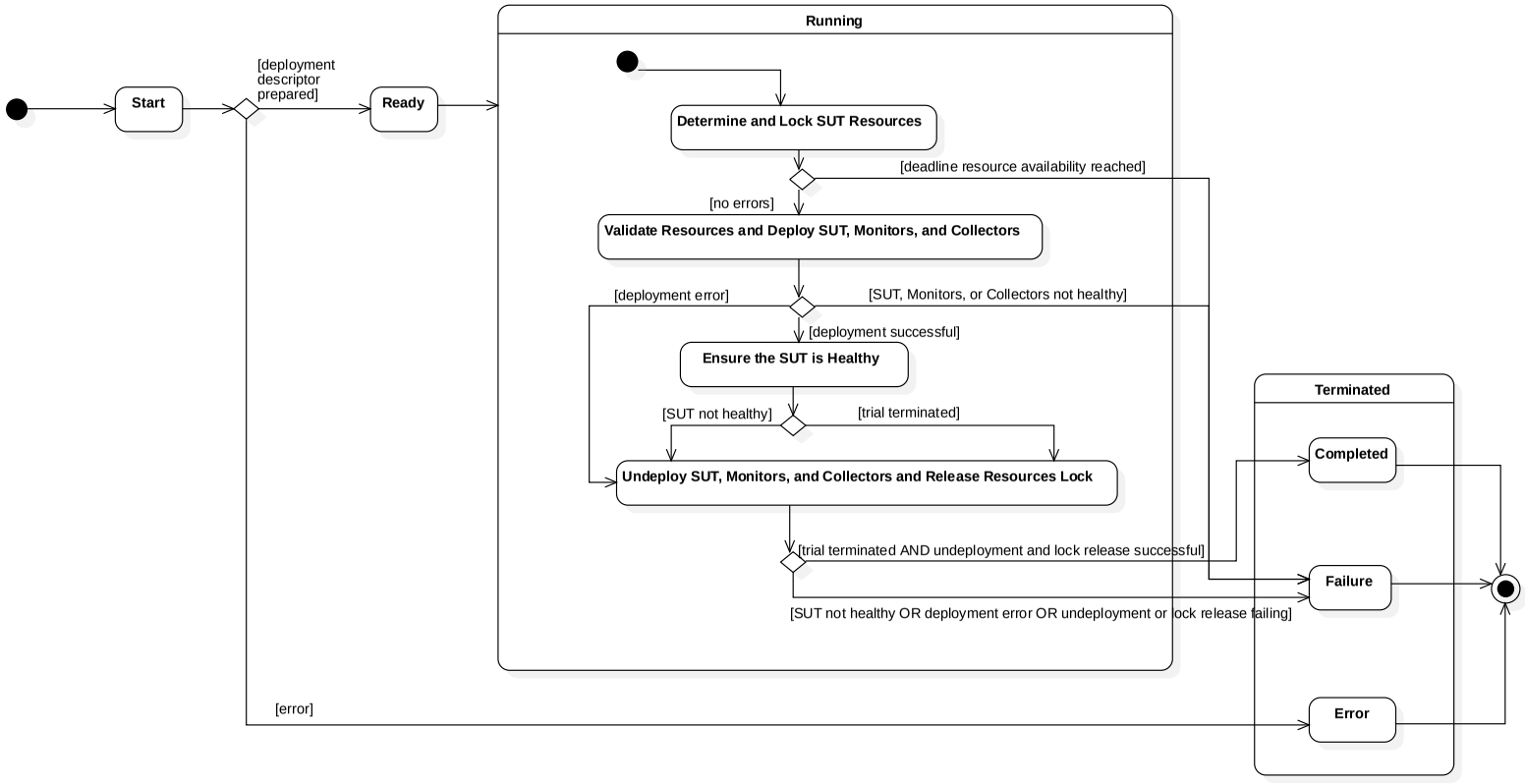

Figure 6.22. BenchFlow Framework: Deployment Manager State Machine

`Lock`, reached after the load driver completes the execution, and taking care of undeploying the SUT and related services, as well as releasing the resources lock. The same state is reached also in case the SUT or related services incur errors during deployment.
If no errors incur during the execution, the final `Completed` state is reached. When error occurs in the `Running` state, the `Failure` final state is reached. Errors can occur in the following states: 1) `Determine and Lock SUT Resources`, when trying to lock resources for the deployment, resource unavailability might trigger a deadline resulting in an error in locking resources; 2) `Validate Resource and Deploy SUT, Monitors and Collectors`, when validating the healthiness of monitors and collectors, in case the mentioned services are not healthy after a given deadline an error incurs; 3) `Ensure the SUT is Healthy`, when validating the SUT healthiness, in case the SUT services are not healthy after a given deadline an error incurs; 4) `Undeploy SUT, Monitors and Collectors and Release Resources Lock`, when during undeployment, errors in cleaning up the test infrastructure or in releasing the resources lock occurs. The same state is reached also in case the SUT or related services deployment incurs errors. Errors during undeployment or resources unlock are relevant to be reported to the users because in many cases the users have to directly act to solve them. We make sure they incur rarely, by applying a mechanism allowing BenchFlow to force the cleanup of the test infrastructure and delete the locks in case of repeated failures.
In Fig. 6.23 we present the state machine implementing the data transformation and analysis lifecycle driven by the *analysis manager*. The lifecycle is instantiated or reactivated for each new event published on dedicated queues on *Kafka* for handling data transformation and metrics computation.
The experiment trial execution lifecycle involves the following states:

1) `Start`, test, and experiment specification, as well as data to be processed, are retrieved from *Minio*. In case of errors in retrieving the data, the lifecycle is moved to the `Error` termination state;

2) `Ready`, reached when data to be processed are ready to be scheduled for processing;

3) `Running`, activated for scheduling data transformers and analyzers and managing the interdependencies among them;

4) `Terminated`, reached at the end of the data transformation and data analysis execution.

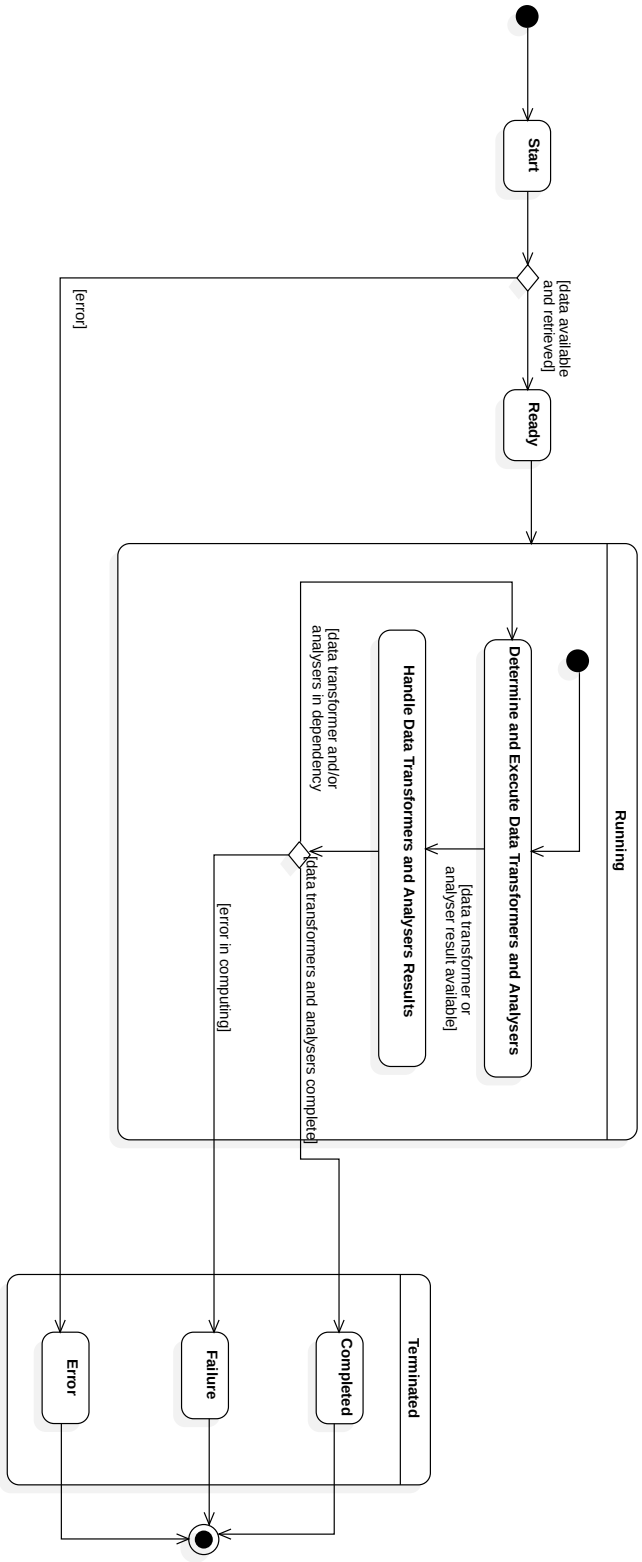

Figure 6.23. BenchFlow Framework: Analysis Manager State Machine

The `Running` state is realized by a sub-state machine, with the following states: 1) `Determine and Execute Data Transformers and Analysers`, determining the data transformers and analyzers to be executed according to the data to be processed. It also builds the dependency graph and handles dependencies among the different metrics and the metrics and the data transformers, by pushing requests for computation on dedicated queues on *Kafka*. The lifecycle waits in this state for dependencies to be available before starting the computation, and checks for their availability at an interval over time; 2) `Handle Data Transformers and Analysers Results`, handling the result and ensuring data transformers and analyzers correctly completed the execution. When other metrics need to be computed, according to the dependency graph, the lifecycle moves back to the `Determine and Execute Data Transformers and Analysers` state.
If no errors incur during the execution, the final `Completed` state is reached. When error occurs in the `Running` state, the `Failure` final state is reached. Errors can occur in the following states: a) `Handle Data Transformers and Analysers Result`, when executing data transformers and analyzers errors might incur, e.g., because of the unexpected format of data to transform.

6.2.5 Performance Data Collection Services

As part of BenchFlow we provide different monitors and collectors services to facilitate data collection activities. They are attached to the SUT services by the *drivers maker* according to the test specification and are deployed using the same standard the SUT relies on for the deployment descriptor. The user is requested to specify only data collection services in the DSL, and we do not distinguish between monitors and collectors. For the user the scope is collecting data, thus she specifies collection services. When monitors are involved and require additional configuration variables, the user is required to specify those variables as part of the collection service configuration in the DSL. Monitor services, monitor the status of the SUT and the underlying infrastructure and retrieve data from them to be evaluated, and are identified by a unique name across monitors and collectors. Collectors, as the name state, collects data from the SUT and the underlying infrastructure. Collectors read and save the data on their file system as temporary files, which are then compressed using gzip[1], and sent to the Minio file storage system.
Both monitors and collectors can fall under one of two categories, namely

[1] http://www.gzip.org/, last visited on February 7, 2021

offline and **online**. An offline monitor needs to be queried for being activated. When queried, it retrieves the monitored data according to its business logic and returns it to the caller. An example of such a monitor is one executing queries on a DBMS. An online monitor on the other end monitors data in real-time and can be queried to retrieve a sample of the monitored data at a given point in time. One example of such a monitor is monitoring the CPU utilization of a service.
Collectors work similarly to the monitors. Offline collectors are queried once to perform a single data collection. One example of such a collector is the one performing a dump of database data. Online collectors collect data from a real-time stream of data and store them on a local buffer. One example of such a collector is one collecting CPU utilization of a service.
The monitor services we provide are:

a) MySQL Monitor (Offline): it allows the user to perform a database query, and compare the result with a provided value. The MySQL Monitor requires the following configuration data: "query", "value" and "method" to provide the query to perform on the database, an expected value, and the comparison method respectively. The comparison method can be "equal" and "nequal" for equality and non-equality of the result. When called the monitor returns a JSON object reporting the response obtained from executing the "query" on the database, and the result of the comparison, that is usually either `true` or `false`;

b) CPU Monitor (Online): it allows the user to start and stop monitoring the CPU usage of one or more linked named services, specified using the optional configuration data: "select", accepting a comma-separated list of named services. When called the monitor returns the CPU percentage usage structured in a JSON object. The percentage usage is reported in the last 5, 30 and 60 seconds, both the total (`last5`, `last30`, `last60`) and the difference between the last monitored values (`last5D`, `last30D`, `last60D`), as well as the timestamp for each value as obtained from the underlying infrastructure (`last5Time`, `last30Time`, `last60Time`).

The collector services we provide are:

a) MySQL Collector (Offline): its job is to perform a full dump of a given set of MySQL tables, which means saving both the values contained in them as well as the schema and the types of the columns. It also saves the size (in bytes) of the database. All collected data is saved and stored

as a Structured Query Language (SQL) dump. The MySQL collector requires the following configuration data: access details to the database, and tables to collect;

b) Stats Collector (Online): its purpose is actively collecting usage data from a given list of services deployed in Docker containers, which primarily include CPU, RAM, DISK, IO, and NET. The usage data are collected by the underlying infrastructure, mostly relying on Docker APIs. Some other data, not exposed by Docker, are directly collected from the infrastructure. One example is NET that in some cases is not provided by Docker [Cerfoglio, 2016, Section 7.2]. In this case, we detect NET are missing and we directly query the underlying infrastructure. The Stats collector requires the following configuration data: list of SUT services to collect data from;

c) Properties Collector (Offline): its function is to collect the properties of a service, as well as the host on which it is running on. A service's properties include its name, ID, available CPU cores, available RAM, and other resource-related data. The host's properties include data similar to the service's properties, such as the total number of CPU cores on the machine, and the total amount of RAM. Similarly to the stats collector, most of the data are obtained via the Docker APIs. The Properties collector requires the following configuration data: list of SUT services to collect data from. This collector is always executed as part of test execution with BenchFlow;

d) Logs Collector (Offline): its scope is to collect the logs (from standard output and standard error) of a service. Similarly to the stats collector, most of the data are obtained via the Docker APIs. The Logs collector requires the following configuration data: list of SUT services to collect data from. Optionally the user can also specify a timestamp after which point in time to collect the logs, relatively to the test start;

e) Zip Collector (Offline): its purpose is to zip a list of provided directories from the file system of given services. It relies on Docker capabilities to mount the volumes from services of interest. The Zip collector requires the following configuration data: list of SUT services, along with a given list of volumes to zip data from;

f) Faban Collector (Offline): its scope is to collect trial execution data from Faban. It is directly integrated as part of the BenchFlow *Faban Driver*

and interacts with the Faban APIs;

g) JMeter Collector (Offline): its scope is to collect trial execution data from JMeter. It is directly integrated as part of the BenchFlow *JMeter Driver* and interacts with the JMeter APIs.

6.2.6 Metrics Computation Framework

Once the data is stored on Minio by a collector, the collector sends a message on a dedicated queue on Kafka signaling the availability of new data to be processed. The message signaling the availability of data always contains also Identifiers (IDs) to identify the test, experiment, and trial to which the data belong to. The analysis manager then takes care of scheduling data transformers and analyzers on Spark, if the data are of interest for computing performance metrics. Spark is designed to scale with the amount of data, and we rely on this framework because performance data to be processed increase in size with the duration of the test, and according to the SUT type. When we first start our work, we targeted WfMSs as SUT, and for computing performance metrics for such a system, database dumps have to be analyzed and those grow fast with the amount of load issued to the SUT [Ferme et al., 2015].
In Fig. 6.24 we present the architecture of Spark, in its distributed deployment configuration. The *Cluster Manager* is the service dedicated to acquiring the resources for the computation from the underlying deployment infrastructure. The resources are made available on *Worker Node*s. The actual business logic of the functions running on Spark is managed by a *Spark Driver* in a *Spark Context*. The Spark driver is controlled by a *Spark Application* representing the business logic of the function, as well as its configuration for being executed on Spark. The Spark driver is executed on worker nodes in an *Executor*, handling one or more Spark drivers (*Task*s).
Spark allows us to define data transformation and data analysis procedure in different programming languages, and facilitates the processing of heterogeneous and unstructured data, like the one collected during performance testing from different SUT types and services.

[2]Source: https://bigdatapath.wordpress.com/2018/05/16/apache-spark-architecture/, last visited on February 7, 2021

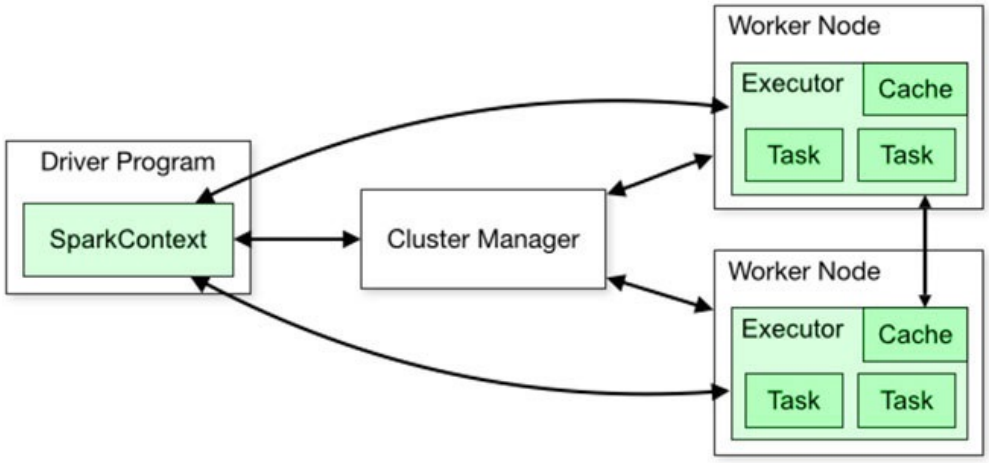

Figure 6.24. Apache Spark Architecture[2]

6.2.7 Trial Execution Frameworks

For the performance test execution, we rely on two state-of-the-art open-source frameworks, Faban and JMeter. We integrate with such frameworks relying on software libraries we develop as part of the BenchFlow framework.
Faban is an open-source tool for performance testing widely used in benchmarks released by the SPEC. Faban is designed for reliable performance test execution and provides scheduling and reporting capabilities. Users can interact with Faban via the provided Web interface.
In Fig. 6.25 we report an overview of then Faban architecture. Faban names a performance test as *Benchmark*. The framework is designed to support distributed load from multiple servers (*agents*) simulating users interacting with the SUT. The *Harness* is the control service providing a queue for benchmarks to be executed and taking care of scheduling the benchmarks by distributing them across all the available agents. The list of available agents is always kept up to date in a *registry*, where the agent registers itself when it starts on a remote server, and the harness retrieves before scheduling a new benchmark. The agents execute the benchmark and issue load to the SUT by executing the load driver behavior relying on multiple *threads*.
JMeter is a widely-used open-source performance testing framework. JMeter is designed to support a wide range of different target protocols, a rich definition

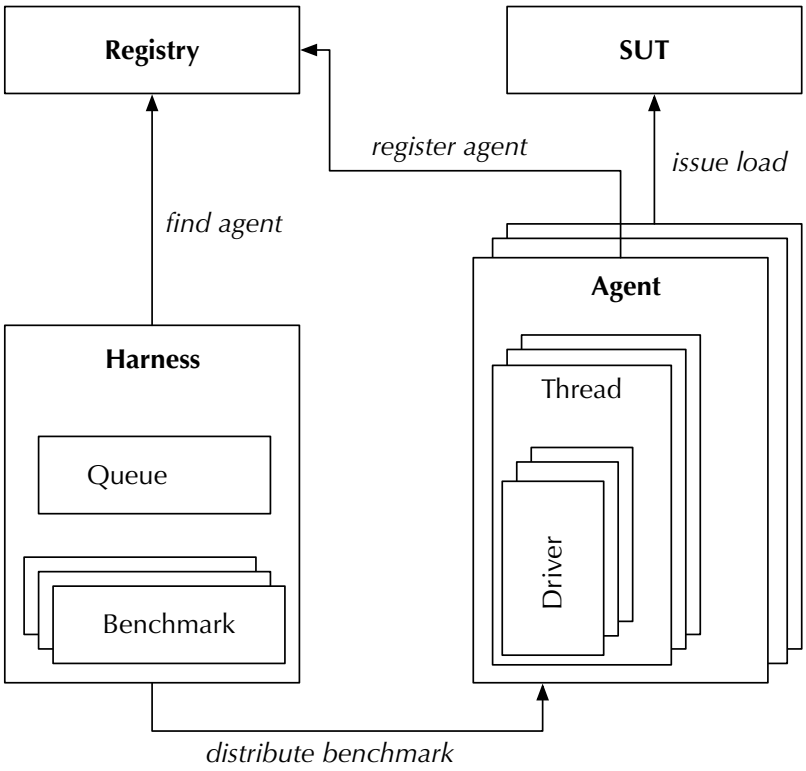

Figure 6.25. Faban Architecture

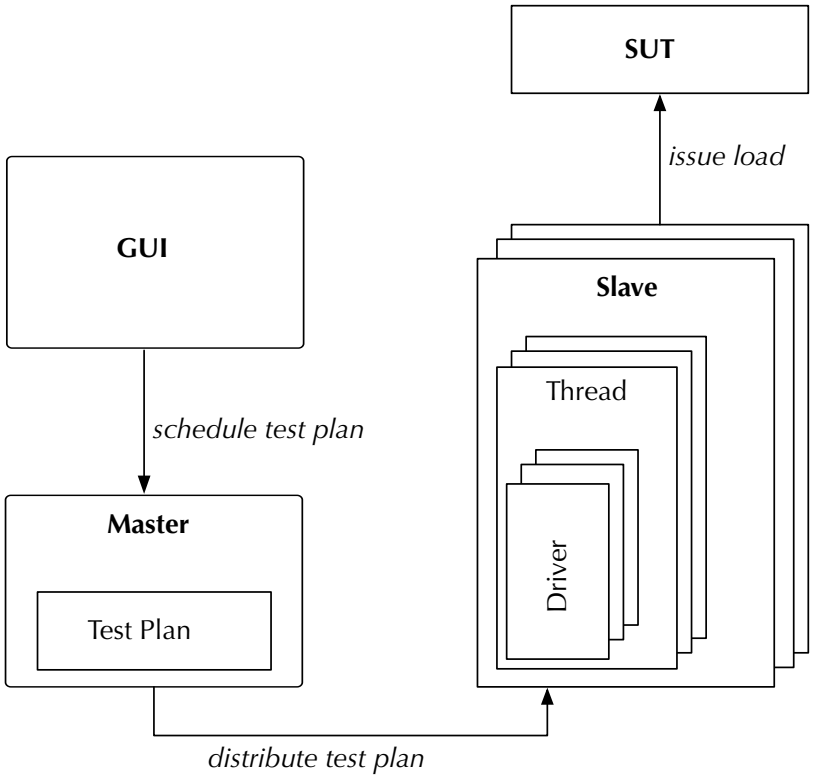

Figure 6.26. JMeter Architecture

of the performance test characteristics and it is extensible employing plugins. Compared to Faban, it offers more advanced features for workload specification, and greater target protocols support. On the other end, it misses the scheduling capabilities and it is not natively designed for distributed performance testing, although it supports the possibility to execute distributed load.
In Fig. 6.26 we report an overview of the JMeter architecture when configured for supporting a distributed load. JMeter names a performance test as *Test Plan*. The framework supports test definition through a GUI, which is also used to schedule the test plan for execution. The execution of the test plan is controlled by the *Manager*, distributing the test plan on different remote servers, called *Slaves*, where *load drivers* issue the load to the SUT relying on multiple executed *threads*. The list of slaves is static and has to be provided as part of the test plan using a list of IPs

6.2.8 Construction of the Bundles and Load Drivers

The user submits one or more tests relying on test bundles and test suite bundles. A test bundle contains the test specification relying on the BenchFlow DSL, the SUT deployment descriptor and test data. A test suite bundle collects one or more test bundles in a single package and additionally contains a test suite specification relying on the BenchFlow DSL. Test and test suite bundles are built starting from artifacts the user provides, and when she relies on the provided CLI, BenchFlow takes care of reading the files in the user's file system and package the test and test suite bundle to be submitted to *test manager*. The test bundle is used as the base for generating experiment bundles by the *test manager*. The experiment bundle is used as the starting point for constructing the actual executable performance test executed by one of the two supported trial execution frameworks, namely Faban and JMeter. The executable performance test is represented by load drivers. The technologies used for constructing the load drivers vary according to the target trial execution framework, and in particular, are based on Java for Faban and on XML for JMeter. Independently from the target framework, the load driver construction is realized by the following steps:

1) parsing of the experiment specification. The experiment specification is already validated as correct and complete;

2) loading of the load driver template according to the experiment specification and the target SUT type. To facilitate the generation of load drivers we provide base templates for different experiment specifications we generate for different performance test goals and supported SUT types, to be parameterized according to the user specification;

3) parameterization of the load driver template and generation of the test related load driver;

4) validation of the generated load driver.

From a single experiment specification, all the load drivers for all the trials are generated at once, so they are ready to be scheduled for execution. The load driver templates we provide also codify best practices based on the literature and our own experience, to optimize the resource utilization of the performance test execution infrastructure and guarantee the correct behavior of the performance test execution. They are built to simplify the load driver construction for both of the main target SUT types of our approach, namely BPMN 2.0 WfMSs and RESTful Web services.

For more details on the technical challenges in constructing load drivers for Faban refer to [Findahl, 2017; D'Avico, 2016], while for the technical challenges in building load drivers for JMeter refer to [Palenga, 2018].

6.3 BenchFlow Framework Implementation

In this section, we discuss the main implementation details of the BenchFlow framework. We present an overview of services' technologies we rely on for building the different services realizing the framework. We then present the deployment diagram of the BenchFlow framework, and an overview of the design of the main APIs utilized for communications among BenchFlow services and for enabling integration with third-party services. We also provide implementation details on the data collection services we include as part of the BenchFlow framework, and all the computed metrics and statistics, and performed statistical tests. In the last part of the section we offer an overview of tools we provide for users' to interact with the BenchFlow framework and to integrate the framework with DevOps tools and CSDL, other than discussing the extension points we make available.

6.3.1 Services' Technologies

The BenchFlow framework is realized by multiple services implemented with different technologies. The core services on the BenchFlow framework are implemented as RESTful Web services. The main technologies and programming languages we rely on are:

a) Java for the core BenchFlow framework services, libraries used by core services and part of the DSL library;

b) Scala for part of the DSL library;

c) Go[3] for monitors and collectors services and libraries used by the same services, due to the possibility to compile Go Lang to a native executable and reducing the resource footprint of services;

d) Python[4] for data transformers and analyzers, and libraries used by them;

e) DropWizard[5] as the base framework for building Java Web services;

[3] https://golang.org/, last visited on February 7, 2021
[4] https://www.python.org/, last visited on February 7, 2021
[5] https://www.dropwizard.io/, last visited on February 7, 2021

f) Apache Spark as the runtime framework for data transformers and analyzers;

g) MongoDB[6] as the database to store the state of the core BenchFlow services;

h) Docker as the packaging and deployment technology for all the services.

To guarantee the implemented services and components of the BenchFlow framework work according to the requirements, we always developed at least unit, integration, and system test for all the services. For the main BenchFlow framework usage scenarios we also developed end-to-end testing and performance testing, relying on the BenchFlow framework itself.
For more implementation details for the different services and components realizing the BenchFlow framework refer to: [Findahl, 2017; D'Avico, 2016; Palenga, 2018] for the core services and the DSL library, and to [Cerfoglio, 2016] for the services related to data collection, data transformation, and data analysis.

State Machine Implementation

For implementing the state machines we relied on a solid open-source library dedicated to the scope. The software library is named Squirrel[7], and provides all the important features for implementing state machine models and instantiating state machine instances when needed. We relied on the mentioned software library for:

a) defining the state machines presented in Sect. 6.2.4. Squirrel supports for hierarchical state specification;

b) instantiate state machine instances for different needs;

c) automatic handling of state transition exceptions.

6.3.2 Deployment Diagram

In Fig. 6.27 we present the deployment diagram of BenchFlow framework core services. The SUT deployment environment, the trial execution framework and load drivers deployment environment, and the BenchFlow core services deployment environment are well-separated to avoid interference and connected over

[6] https://www.mongodb.com/, last visited on February 7, 2021
[7] https://github.com/hekailiang/squirrel, last visited on February 7, 2021

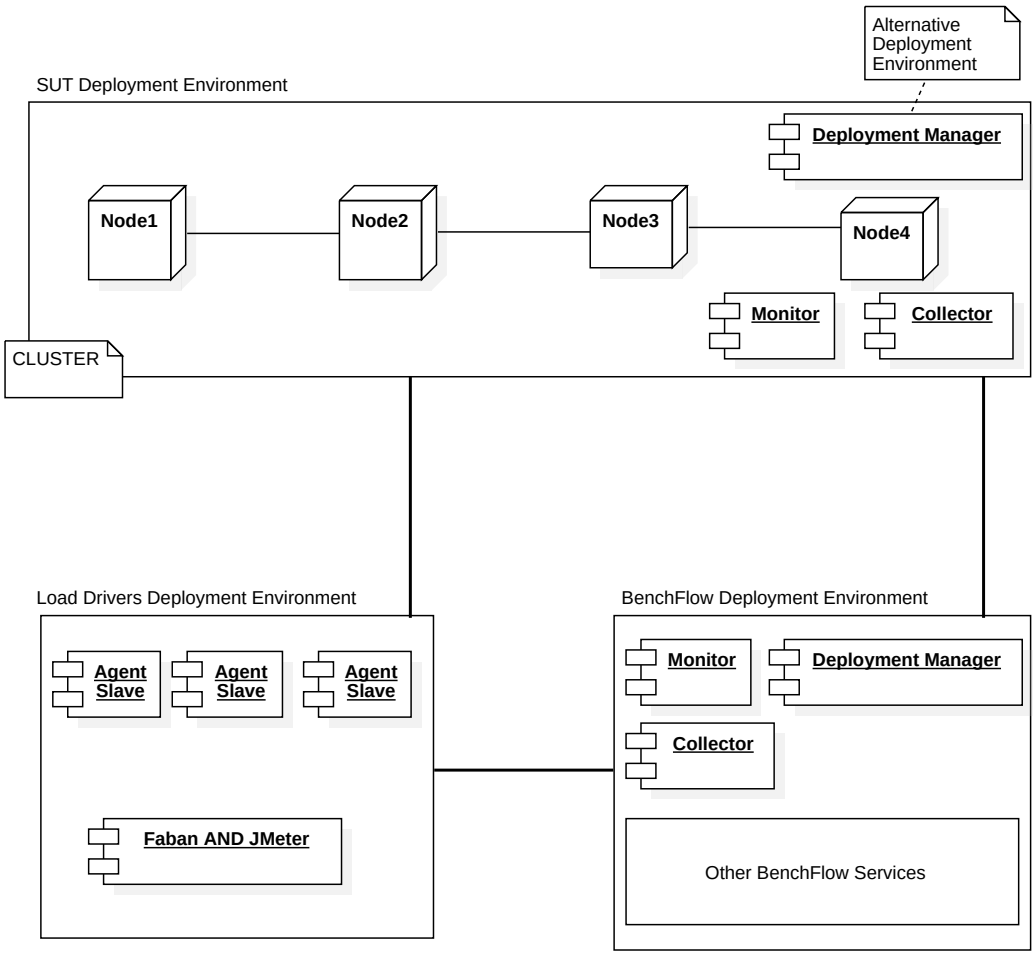

Figure 6.27. BenchFlow Framework: Deployment Diagram

the internet (possibly over Ethernet in a local area network). Connecting the deployment environment in a local area network maximizes the stability of the collected performance data, especially the client-side ones related to load drivers. In some situations, this is not possible to be guaranteed, for example when the different environments reside on different infrastructures. In the SUT deployment environment, other than the SUT services, we also deploy monitors and collectors services and the *deployment manager*. When this is not possible, for example, due to security constraints, monitors and collector services and the *deployment manager* are deployed on the BenchFlow core services deployment environment although some overhead in the data collection and SUT deployment operations are introduced. In the trial execution framework and load drivers deployment environment, we deploy the trial execution framework main services, thus Faban and JMeter, as well as deploy the Agents/Slaves to execute the load drivers in a distributed manner. The BenchFlow core services deployment environment hosts the core services supporting the declarative performance test automation.
Services related to the *Analysis* phase are usually deployed on separated hardware, sized according to the needs of the analysis framework, in our case Apache Spark.

6.3.3 Main Services' APIs Implementation Details

BenchFlow services expose APIs for internal communications and for enabling third-party services integration. The APIs are designed following the REST standard, and in particular, following the recommendations of the article on Resource Naming by T. Fredrich [Fredrich, Todd, 2012]. Recommendations in [Fredrich, Todd, 2012] suggest to define REST resources to represent entities in the domain model of the service, and that the Uniform Resource Identifier (URI) over which the APIs is exposed should describe the hierarchical relationships between the resources representing the different entities of the service domain model.
Across the different services we define a set of resources that are always available:

a) **users**, representing the user interacting with the service;

b) **tests**, representing a test specification;

c) **experiments**, representing an experiment specification generated from a test;

d) **trials**, representing a trial of an experiment.

The presented entities are closely related to one another in a hierarchical structure, with the base entity being the user.
To guarantee the different services can always identify the resource of interest while communicating, we defined a correlation ID for each of the main resources listed above. The ID is constructed such that it respects the same recommendations of the APIs naming, thus it is structured hierarchically to always being able to correlate all the related entities. The structure of the IDs for the different resource are:

a) **`{testID}`** = `{username}.{testName}.{testNumber}` for the test resource;

b) **`{experimentID}`** = `{testID}.{experimentNumber}` for the experiment resource;

c) **`{trialID}`** = `{experimentID}.{trialNumber}` for the trial resource.

The *testID* always reports the username of the user the test belongs to. The *testName* represents the name the user assigned to the test and the *testNumber* is a sequential number added by the *test manager* in case of multiple tests with the same name are submitted by the same user. An example of *testID* would be `benchflow.loadTest.1`. The *experimentID* references to the *testID* and adds an incremental number automatically generated by the framework, to guarantee the ID is unique. An example of ID for the first experiment of the example *testID* would be `benchflow.loadTest.1.1`. The *trialID* references to the *experimentID* and as per the experiment, we add an incremental ID. An example of ID for the first trial of the example *experimentID* would be `benchflow.loadTest.1.1.1`.
For each resource, we define APIs to create the resource (*POST*), to read the resource (*GET*), to update the resource (*PUT* and *PATCH*), and optionally we provide APIs to delete the resource (*DELETE*). The APIs are designed to respond back with data serialized in JSON format. The response always contains the complete model of the response entity and all the relevant information (i.e., the resource URL) to navigate from any given entity to related entities of interest. The APIs response code are designed as such to respect the HTTP response code standard[8], and to provide the correct information to the user according to the internal state of the resource after a request.

[8] https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html, last visited on February 7, 2021

The complete implementation details for the BenchFlow services' APIs are provided in the following Master theses [Findahl, 2017; Palenga, 2018; D'Avico, 2016; Cerfoglio, 2016].

6.3.4 Performance Data Collection Services

All the monitor and collector services we introduce in Sect. 6.2.5, but the ones dedicated to Faban and JMeter, are implemented using the Go programming language and packaged in Docker Images along with all their dependencies. We opted for the Go programming language because it is a lightweight and efficient language, and can be compiled to a native binary file. It is important for monitors and collectors to be lightweight and with limited dependencies to the underlying operating system because they are usually deployed next to the SUT services and interference on the performance of the SUT must be avoided during the performance test execution. Collectors are designed to optimize the time they require to collect data so that they impact as little as possible the SUT performance and the trial execution time. Since all the services rely on the same functionalities to interact with the BenchFlow services, e.g., zipping data, sending data on Minio, signal Kafka, we provide a library for the mentioned functionalities, to be reused for monitors and collectors. Faban and JMeter collectors are directly integrated with their respective BenchFlow drivers.
The MySQL collector service depends on monitors execution before being able to offer their functionalities. It first depends on the *CPU* monitor, to determine when the service on which the MySQL collector is attached has low resource utilization, and then the *MySQL* monitor to execute queries on specific tables on a database to determine when it is the right time to collect a dump of the database. The trial execution lifecycle takes the dependencies into account, for correctly starting and stopping monitors and collectors in the correct ordering. According to the *phase* in which the monitor is defined to be started, the trial execution lifecycle: a) completes the interaction with the monitor, before starting the collector, in case of the *start* or *all* phase; b) completes the interaction with the monitor, before stopping the collector, in case of the *end* or *all* phase; The trial execution lifecycle accounts for possible errors, retrying the operations until a given retry threshold is reached.
Monitors and collectors can be executed in different moments of the trial execution lifecycle, as anticipated in Sect. 6.2.4. In the following, we report the phase in which each of the provided monitors and collectors is executed. For monitors:

a) MySQL Monitor (Offline) phase: *end*

b) CPU Monitor (Online) phase: *all*

For collectors:

a) MySQL Collector (Offline) phase: *end*

b) Stats Collector (Online) phase: *all*

c) Properties Collector (Offline) phase: *end*

d) Logs Collector (Offline) phase: *all*

e) Zip Collector (Offline) phase: *end*

f) Faban Collector (Offline) phase: *end*

g) JMeter Collector (Offline) phase: *end*

All monitors and collectors provide a uniform REST APIs for obtaining data from the monitors and storing data with the collectors, as well as starting/stopping monitoring/collection for the online ones. The APIs depend on whether the monitor or collector is an online or offline one. Offline monitors expose the following APIs and methods:

a) `GET /data`, returning the data in JSON format, according to the called collector. The APIs is used for getting data of interest for monitoring the state of the SUT.

Online monitors add the following APIs and methods to the ones exposed by the offline monitors:

a) `POST /start`, used for requesting the monitor to start monitoring data.

b) `DELETE /stop`, used for requesting the monitor to stop monitoring data.

Offline collectors expose the following APIs and methods:

a) `PUT /store`, used for requesting the collector to store the collected data on *Minio* and signal *Kafka*. For an offline collector, the actual collection happens when this method is called. For online collectors, the data is collected over time.

Online collectors add the following APIs and methods to the ones exposed by the offline collectors:

a) `POST /start`, used for requesting the collector to start collecting data.

b) `DELETE /stop`, used for requesting the collector to stop collecting data.

The monitors and collectors respond with a $HTTP200$ response if successful, and if not with an appropriate error response, such as $HTTP405$ for invalid methods when calling the API. For some monitors and collectors, additional data has to be provided. Such data is passed either as parameters or in the body of the request, according to the $HTTP$ method used to interact with the service. To handle such data, we rely on a custom library built as part of BenchFlow, to correctly integrate the monitors and collectors with the trial execution lifecycle. By providing a uniform APIs we provide a standard that all monitors and collectors comply with, thus making it easy to plug in new monitors and collectors into the system without having to modify other components.

```
{ String : <unique name of the monitor/collector> }
    <SUT DEPLOYMENT DESCRIPTOR>

endpoints:
    start: { String : <endpoint offering the start functionality> }
    stop: { String : <endpoint offering the stop functionality> }
    # OPTIONAL IF monitor
    data: { String : <endpoint offering the data functionality> }
    # OPTIONAL IF collector
    store: { String : <endpoint offering the store functionality> }

phase: { String : < "start"  "end"  "all" > }

# OPTIONAL
dependencies:
    # OPTIONAL
    monitors:
        - { String : <name of a monitor or a collector> }
        ...
    # OPTIONAL
    collectors:
        - { String : <name of a monitor or a collector> }
        ...
```

Listing 6.2. The Monitors/Collectors YAML Format Specification

Monitors and collectors services, but the ones dedicated to Faban and JMeter, are deployed using the same deployment descriptor formalism as per the SUT as presented in Sect. 5.3.3 (line #2). The actual deployment descriptor is enriched for including data about the exposed APIs endpoints (lines #4-#10), the *phase* in which the service execute (line #12), the dependencies to other monitors or collector (lines #15-#23), and BenchFlow specific variables provided to facilitate the automation. In Listing 6.2 we report the YAML specification for monitor and collector services.
The list of supported BenchFlow variables, that can be provided as part of the *environment* block in the *<SUT DEPLOYMENT DESCRIPTOR>* is:

a) `BENCHFLOW_TEST_<BENCHFLOW_SERVICE_NAME>_<BENCHFLOW_SERVICE_INFO>`: refers to data of the specified BenchFlow service;

b) `BENCHFLOW_TEST_<BENCHFLOW_SERVICE_NAME>_<ENVIRONMENT_VARIABLE_NAME>`: refers to configuration variables, part of the *environment* section of the specified BenchFlow service;

c) `BENCHFLOW_TEST_BOUNDSERVICE_<BOUNDSERVICE_SERVICE_INFO>`: refers to data retrieved from the SUT service the monitor or collector is attached to. When the monitor or the collector is attached to more than one service the variable resolve to the list of <BOUNDSERVICE_SERVICE_INFO> for all the bound services, mapping each data to the corresponding service;

d) `BENCHFLOW_TEST_BOUNDSERVICE_CONFIG_<ENVIRONMENT_VARIABLE_NAME>`: refers to configuration variables, part of the *environment* section of the SUT service the monitor or collector is attached to. When the monitor or the collector is attached to more than one service the variable resolve to the list of <ENVIRONMENT_VARIABLE_NAME> for all the bound services, mapping each data to the corresponding service;

e) `BENCHFLOW_TEST_CONFIG_<ENVIRONMENT_VARIABLE_NAME>`: refers to collection services configuration variables, part of the *data collection* section of the test specification.

where:

a) `<BENCHFLOW_SERVICE_NAME>`: is one of *KAFKA*, *MINIO*;

b) `<BENCHFLOW_SERVICE_INFO>`: is one of *IP*, *PORT*;

c) `<BOUNDSERVICE_SERVICE_INFO>`: is one of *IP*, *PORT*, *VOLUMES*, *CONTAINER_NAME*. *VOLUMES* represent the data volume of the service, and *CONTAINER_NAME* refers to the name of the service;

d) `<ENVIRONMENT_VARIABLE_NAME>`: is the name of one environment variable part of the referenced service deployment descriptor, either a BenchFlow service or a bound service, or specified in the test specification submitted by the user as part of the *data collection* configuration.

All the variable types are prefixed with `BENCHFLOW_TEST_`, to avoid collisions with other environment variables used for configuring the SUT and the substitution mechanism for variables implemented as part of Docker. To determine variables resolution order, BenchFlow builds a graph where each vertex is a variable, and each edge is a dependency between variables. In case the graph has cycles, BenchFlow will report an error; otherwise, the vertexes of the graph will be traversed in topological order, and the variables will be resolved to their actual value.
In the following, we report the expected configuration variables for each of the implemented services. For monitors:

a) MySQL Monitor (Offline): the name of the database to monitor (`MYSQL_DB_NAME`), the name of the tables of interest (`TABLE_NAMES`) the IP (`MYSQL_IP`) and port (`MYSQL_PORT`) to read the DBMS, a username (`MYSQL_USER`), a password (`MYSQL_USER_PASSWORD`) to access the monitored database, with the correct permissions, the query to perform on the database (`COMPLETION_QUERY`), an expected return value from the query (`COMPLETION_QUERY_VALUE`), and the method for comparison (`COMPLETION_QUERY_METHOD`);

b) CPU Monitor (Online): the list of services (`SERVICES`) to monitor.

For collectors:

a) MySQL Collector (Offline): same as the MySQL monitor;

b) Stats Collector (Online): the list of services (`SERVICES`) to collect stats from;

c) Properties Collector (Offline): the list of services (`SERVICES`) to collect proprieties from;

d) Logs Collector (Offline): the list of services (`SERVICES`) to collect logs from, an optional timestamp after which point in time to collect the logs (`START_FROM`), relatively to the test start;

e) Zip Collector (Offline): the list of services (`SERVICES`) along the list of volumes to be zipped (`TO_ZIP`);

f) Faban Collector (Offline): the ID of the executed trial;

g) JMeter Collector (Offline): the ID of the executed trial;

When a variable is mandatory it has to be provided either via the APIs or as an environment variable. When both options are used, the ones provided via the APIs have priority.
All the monitors and collectors also implicitly specify the following variables as part of their deployment descriptors:

a) `<BENCHFLOW_DATA_NAME>`: the unique name to assign to the data, and the queue to send data on Kafka for collectors. It is usually the same as the name of the service;

b) `<BENCHFLOW_MINIO_IP>`: the IP of the Minio service;

c) `<BENCHFLOW_MINIO_PORT>`: the port of the Minio service;

d) `<BENCHFLOW_MINIO_ACCESSKEYID>`: the access key of the Minio service;

e) `<BENCHFLOW_MINIO_SECRETACCESSKEY>`: the secret access key of the Minio service;

f) `<BENCHFLOW_KAFKA_IP>`: the IP of the Kafka service;

g) `<BENCHFLOW_KAFKA_PORT>`: the port of the Kafka service.

Monitors and collectors can be extended over time, according to BenchFlow evolution and requirements. They all implement the mentioned interfaces and can be integrated as part of the BenchFlow framework. An in-depth discussion of how the deployment descriptors for each of the monitors and collectors are defined, refer to [Cerfoglio, 2016].

6.3.5 Data Transformation and Metrics Computation Functions

Data transformation (i.e., data analyzers) and metrics computation (i.e., analyzers) functions are defined as *Spark Applications*. The collected data are then analyzed by relying on applications written on top of Apache Spark, and transformed data as well as computed metrics and statistics, are stored on Cassandra. Data transformers and analyzers are scheduled by the analysis manager for execution on Spark, according to the defined dependencies.
Different metrics and statistics require different data, whether is raw performance data collected from the SUT or other metrics or statistics. For example, before computing statistics related to the resource usage of SUT services, the average CPU percentage usage, we need the raw performance data of the CPU usage overtime to be available. It is also possible for some aggregate metrics and statistics to depend on multiple others, and thus the analysis scheduler needs to wait for such data to be available before proceeding with the computation. Similarly, experiment statistics depend on the trial metrics related to the experiment, and thus they can not be processed unless all trials have been processed. We specify the dependencies among data availability and metrics at different levels, by providing a configuration file to the analysis manager. The analysis scheduler, when scheduling data transformers and analyzers take into account the specified dependencies and schedule the work according to them, prioritizing the data transformation and analysis of data for computing the metrics referenced in the test specification (usually in the *Observe*, *termination_criteria* and *quality_gates* sections) when data are available. Prioritizing metrics referenced in the test specification allows us to provide back useful data to the *Goal Exploration* as soon as possible, so decisions can be taken. More detail on this can be found in [Cerfoglio, 2016].

Catalog of Available Performance Metrics, Statistics, and Statistical Tests

The users of BenchFlow can specify tests and test suites. Each test executes one or more experiments, and each experiment executes one or more trials. Performance data are collected for each executed trials, and metrics and statistics are defined and computed for each trial and then for the experiment taking into account the metrics of all the trials. Collected data refer to the entire duration of the load functions, including ramp-up, steady-state, and ramp down. Metrics are computed for the entire duration of the load function, as well as distinguishing the different moments of the load function. The user of the DSL

refers to metrics computed on the steady-state. The other ones are provided for analysis purposes.
We define many different metrics and statistics, to accommodate different requirements of BenchFlow users, as well as important statistical tests in the context of performance data analysis. Metrics are defined on the client (*Workload* in the DSL) side and on the SUT (*services* in the DSL) side. A consistent set of metrics are related to BPMN 2.0 WfMSs, being a SUT type largely tested with BenchFlow. BPMN 2.0 WfMSs usually stores performance data in DBMS, and those data are related to the internal behavior of how this type of system works. Each BPMN 2.0 process model can be instantiated in different *process instances*, and each process instance has its lifecycle, starting at a given time and ending after its execution. Each process instance executes one or more BPMN 2.0 constructs (e.g., an activity or a gateway), each having its lifecycle and thus a starting and an ending time. A comprehensive description of metrics is provided in [Ferme et al., 2019].
In the following, we report all the metrics, descriptive statistics, and statistical tests we implement and make available to the users. We report the name the user can use to refer to the metric in parenthesis. We also report when the defined metrics are specific to BPMN 2.0 WfMSs.

Metrics We provide the following metrics on the client-side, applicable to all the SUT types:

1) Total Operations (*n_operations*): counts the total number of performed operations, which in our context represents interactions between the client and the SUT, meaning the load generated for the test. Important to measure the total number of issued calls to the SUT;

2) Successes/Failures Operations (*n_operation_successes*/*n_operation_failures*): counts the successes and failures of the operations. Important to know about the results of the interactions to the SUT;

3) Mix Deviation (*mix_deviation*): a mix represents the policy used to select operations when simulating the workload, and each operation can have a mix value representing the ratio they are performed with. This metric measures the deviation from the mix specified in the test definition happened during the execution of the experiment;

4) Think Time Deviation (*think_time*): measures the deviation from the think time specified by the user in the test definition, happened during the execution of the experiment. It is computed for each mix of operations;

5) Response Times (*response_time*): measures the response time of the SUT to the client after operating, expressed for example in nanoseconds;

6) Latency (*latency*): measures the latency of the SUT to the client when operating, expressed for example in nanoseconds;

7) Little's Law (*little_law*): the Little's law formula by [Little, 1961] states that the average number of users in a system is equal to the arrival rate multiplied by the average time a user spends in the system ($users = arrival\_rate * time\_spent$). This law allows us to actively compare the number of users defined for a test versus the number of users that were simulated by the load driver.

We provide the following metrics on the server-side, applicable to all the SUT types:

1) CPU Percent Usage per Second (*cpu_perc_usage_sec* or *cpu*): the percent usage of the CPU each second;

2) RAM Usage per Seconds (*ram_usage_sec* or *memory_usage_sec* or *ram*): the total usage of the RAM on the system each second;

3) IO Operations per Seconds (*io_n_operations_sec*): the total number of IO operations on the system each second;

4) DISK Read Operations (*disk_n_read_operations*): the total number of DISK read operations on the system each second, both synchronous and asynchronous;

5) DISK Write Operations (*disk_n_write_operations*): the total number of DISK write operations on the system each second, both synchronous and asynchronous;

6) DISK Total Operations (*disk_n_operations*): the total number of DISK operations on the system each second, both synchronous and asynchronous;

7) NET Percent Usage per Seconds (*net_perc_usage_sec*): the percent usage of the NET each second;

8) Received NET Data (*net_n_received_data*): the amount of data, represented as packets and as bytes, received by the system;

9) Sent NET Data ($net\_n\_sent\_data$): the amount of data, represented as packets and as bytes, sent by the system;

10) NET Errors (net_n_errors): the number of error packets over the network;

11) Resource Cost ($resource\_cost$): the cost of utilized resources (*CPU*, *RAM*, *DISK*, and *NET*). The cost is computed in dollars ($) by multiplying the unit cost for each resource for the actual utilization units of the given resource, and then summing up together the cost of the single resources to provide a single cost value. The user can specify the unit cost of the resources in the configuration options of the framework developed as part of the proposed approach.

We provide the following custom metrics on the server-side, for BPMN 2.0 WfMSs:

1) Execution Time ($wfms\_execution\_time$): the total time to execute the workload of the system, defined as $wfms\_execution\_time = max(wfms\_process\_instance\_end\_time) - min(wfms\_process\_instance\_start\_time)$, the difference between the largest end time of all executed processes and their smallest start time;

2) Throughput ($wfms\_throughput$): the ratio between the number of process instances completed per second and the execution time of the workload. This is computed for all processes instances, as well as for process instances of individual BPMN 2.0 models;

3) Databases' Size ($wfms\_databases\_size$): the size in bytes of the databases used by the WfMSs to store process data;

4) Number of Process Instances ($wfms\_n\_process\_instances$ or $wfms\_number\_process\_instances$): the number of process instances completed during the execution of the trial, either for all process instances or for instances belonging to individual BPMN 2.0 models;

5) Process Instances Durations ($wfms\_process\_instance\_duration$ or $wfms\_process\_instance\_execution\_time$): the duration of process instances in the system, for all of them combined, or per individual BPMN 2.0 model. The duration of a process instance is defined as $wfms\_process\_instance\_duration = wfms\_process\_instance\_end\_time - wfms\_process\_instance\_start\_time$;

6) Number of Construct Instances ($wfms\_n\_construct\_instances$): measures the amount of construct instances completed, for all constructs instances and for construct instances of individual construct definitions;

7) Construct Instances Durations ($wfms\_construct\_instance\_duration$): the duration of construct instances in the system, for all the construct definitions across all process instance and per individual construct definition.

Statistics In the following, we define a series of statistics that can be applied to the computed metrics. The computed statistics provide data about the performance metrics to the users and can be used as part of the test specification. We include common statistics used in the performance testing domain, accounting for the non-normal distribution of performance data usually experience [Montgomery and Runger, 2013].
The statistics we compute at the trial level are:

1) Number of Data Points (n_data_points): the number of data points in the given data set, also known as the sample size;

2) Average (avg): the average value of the data points, also called mean;

3) Q1/2/3 ($q1/q2/q3$): the first, second, and third quartile of the data points. The second quartile is also known as the median. Quartiles are points, not always belonging to the data set, dividing the data set into four subsets, each containing a quarter of the data. They are relevant for observing the distribution of the data;

4) P90/95/99 ($p90/p95/p99$): The 90th, 95th, and 99th percentile of the data points. Percentiles divide the data set so that 90%, 95%, and 99% of its points are lower than the respective percentile value. Similarly to quartiles, percentiles allows to examine the distribution of the values in the data set;

5) Percentiles 1-100 ($p1 - 100$): all the percentiles (from the first to the 100th) of the data points;

6) Min/Max (min/max): the minimum and maximum values of the data points;

7) Mode ($mode$): the mode of the data points. The mode represents one or more values who appear the most in the data set;

8) Variance (var): the variance of the values of the data set. The variance indicates how much the values in the data set differ from one another;

9) Standard Deviation (std): the standard deviation of the set of data. The standard deviation is used to represent the scatter of the values in the data set;

10) Confidence Interval ($\alpha = 0.05$)% ($ci95$): the interval of confidence with $\alpha = 0.05$. The confidence interval represents the interval of values in which a certain measurement, in our case the mean, falls with 95% probability;

11) Standard Error ($stderr$): the standard error of the mean of the set of data. Represents the precision in which we know the population means of the set of data. The lower the standard error, the closer the observed mean is to the population mean. It is useful to determine how precise our computed mean is;

12) Margin of Error ($margin\_err$): the margin of error of the mean of the set of data. We compute the margin of error as $margin\_err = stderr * 2$. Similarly to the standard error, the lower the margin of error, the closer the observed mean is to the population mean;

13) Integral ($integ$): the integral of the data set, calculated as a cumulative trapezoid integration of the data set. It can be applied only to data representing values over time, for example, the CPU usage per second. The integral is relevant to compute the efficiency;

14) Relative Efficiency ($relative\_efficiency$): the ratio between the integral of the data set, and its maximum value multiplied by its size. Relative efficiency is primarily used with resource usage data, where we want to know how efficiently a particular resource is used by the SUT. Values close to one indicate the resource is used to near its fullest;

15) Absolute Efficiency ($absolute\_efficiency$): same as relative efficiency, but instead of the maximum value of the data set we use the maximum value that data set could potentially have. For CPU percentage usage, for example, the maximum would be 100%.

At the experiment level, we also define statistics to compute over trials where the previous statistics were already applied, allowing us to select the most relevant statistics computed over the trials. We additionally compute descriptive

statistics allowing the user to aggregate the performance data collected across multiple trials, and evaluate how reliable the collected performance data are. The descriptive statistics we compute at the experiment level are:

1) Min/Max Q1/2/3 ($min\_q1/min\_q2/min\_q3/max\_q1/max\_q2/max\_q3$), Min/Max Average ($min\_avg/max\_avg$), Min/-Max Mode ($min\_mode/max\_mode$), Min/Max Percentiles ($min\_p1 - 100/max\_p1 - 100$), Min/Max Min/Max ($min\_min/min\_max/max\_min/max\_max$), Min/Max Confidence Interval (min_ci95/max_ci95): the minimum and maximum values across all trials for the mentioned statistics, for example the maximum Q1 across all the trials in the experiment they belong to;

2) Weighted Average ($wavg$): the average amongst all trials, weighted by the number of data points in every single trial. This can only be applied where each trial has a variable number of data points;

3) Best/Worst/Average Trials ($best\_trial\_id/worst\_trial\_id/avg\_trial\_id$): the identifier for the best, worst and average trials in the experiment, obtained by comparing them by the mean value for the trial first and margin of error for the trial as second comparison criterion;

4) Coefficient of Variation ($coefficient\_variation$): the ratio between the standard deviation of the means of the trials and the mean of all the trials, expressed as a percentage. This coefficient is primarily used to convey the dispersion of the values of the trials, where a low coefficient of variation implies the values across all trials tend to be stable, while a high coefficient implies they are not stable;

5) Combined Variance ($combined\_var$): also known as the pooled variance, it is a higher precision estimate of the variance for all the trials combined.

Statistical Tests In addition to computing statistics on the data, we also define a statistical test, which is performed over all trials in an experiment, each trial is a data sample for the test.
We compute the following statistical tests:

1) Levene's Test ($levene\_test$): used to test if a given number N of samples have equal variances. Equal variances across samples are defined as homogeneity of variance. It is solid for non-normal data, as it is usually the case for performance data.

The Levene's test is used to test the assumption the data from our trials have equal variances, meaning they are homogeneous[9]. If the trials in an experiment have homogeneous variances, then we have a good indication the data from the trials are comparable, providing validation for the analysis performed across all trials. We compute Levene's test over two or more trials, using as reference the mean, median, and the trimmed mean of the samples. We thus perform the Levene's test three times and produce three values and the relative p-values. If the p-value is below a certain significance level, which is generally set to 0.05, then the null hypothesis of Levene's test (which states the variances are equal) is rejected, and the variances are thus considered non-homogeneous amongst the trials. This is because if the p-value is below the given significance level then the implication is that the differences in variances of the provided samples are not just due to random sampling, and thus we must reject the assumption that the variances are homogeneous. For additional implementation details about metrics computation, refer to [Cerfoglio, 2016]
As part of the metrics and statistics computation to be provided to the *Goal Exploration*, we also compute the value for the mean absolute error ($mean\_absolute\_error$) for the prediction model, when one is built. Additionally we evaluate the regression absolute ($regression\_delta\_absolute$) and regression relative ($regression\_delta\_percent$) indicators.

6.3.6 User-side Tools

For users to interact with the BenchFlow framework we provide a CLI. The CLI is developed to facilitate the interactions with the *test manager* by exposing useful commands to validate test and test suite specification, schedule tests and test suites, get the list of executing tests, abort tests execution, and get test results and artifacts. The CLI also exposes some options to the users, e.g. to select the trial execution framework, to pass endpoints of already deployed SUT, and to enable reuse of performance test specifications. Examples of user experience, and use cases relying on the CLI to interact with the BenchFlow framework are discussed in [Findahl, 2017, Chapter 6]. In [Findahl, 2017, Chapter 6] some APIs are slightly different because they evolved, but the user experience and the interactions are representative of the current version of the framework.
The CLI also simplifies the deployment procedure of the BenchFlow framework by providing facilities and deployment descriptor automation for connecting to

[9] http://www.itl.nist.gov/div898/handbook/eda/section3/eda35a.htm, last visited on February 7, 2021

remote servers and deploying BenchFlow services. The services' deployment descriptors are retrieved from the current BenchFlow framework release, and are defined relying on Docker Compose. The CLI is packaged in Docker as for the other BenchFlow services so that the users can execute it in any context where Docker is supported.
When integrating the BenchFlow framework with third-party solutions, users working on the integration need to access the BenchFlow *test manager* APIs. For this we provide a client library built from the test manager APIs specification, to facilitate the integration. The client library can be provided in multiple programming languages, e.g. Java, Go, and Python.

6.3.7 Integration with DevOps tools and CSDL

We enable the integration with DevOps tools ad CSDL by relying on the same CLI the user can use for interacting with the BenchFlow framework. The CLI can be embedded as part of the workflows and pipelines built with CICD tools and used for scheduling test suites or tests. The CLI facilitate the integration with the CICD tools by collecting all the data about the current SUT context in the pipeline as described in Sect. 4.4, and passing them to the BenchFlow framework alongside the test or test suite bundle. We currently support the events generated by Jenkins[10] when the software is versioned on GitHub[11]. The CLI also allows to abstract of the interactions with the *test manager* APIs when a test suite is involved. A test suite is a collection of performance test specifications submitted as a single bundle. When interacting with the *test manager* the CLI maps the test suite specification to the different scheduled tests so that a state of the execution for all the tests part of a test suite can be retrieved. The CLI also takes care of evaluating the test suite quality gates, by looking at the results of all the tests involved as part of the test suite.

6.3.8 Extensions Points

We design and implement the BenchFlow framework to be open for extensions, both in terms of the provided functionalities and for integration with third-party solutions. We enable extendability of the framework functionalities by providing a CLI to be used by users or by other systems to interact with BenchFlow's services, RESTful APIs for services to interact with each other and for other systems to interact with BenchFlow, mainly with the *test manager*.

[10] https://www.jenkins.io/, last visited on February 7, 2021
[11] https://github.com/, last visited on February 7, 2021

The APIs are also accessible via automatically generated client libraries, we generate for each service to facilitate the interaction among BenchFlow services and to ease the integration of third-party services with the *test manager*. The client libraries are currently generated in Java and Go, relying on Swagger[12]. Other languages are supported by design when needed, relying on Swagger Codegen[13] capabilities.
The computed metrics are open for the extension as well, by relying on the computational capabilities of Apache Spark.

6.4 Concluding Remarks

In this chapter we present the BenchFlow framework we propose for automating declarative performance tests defined relying on the DSL presented in Chap. 5. We illustrate the BenchFlow architecture, we discuss the main services and their responsibilities and present their interactions and the business logic of core services represented and implemented using state machines. The execution logic of BenchFlow is defined so it can be programmed using the DSL. We also present all the metrics, statistics, and statistical tests we compute on collected data and provide to the users to be integrated with the DSL, as well as for further in-depth performance analysis. We then present the main RESTful APIs and the CLI we provide for interacting with BenchFlow and for integration with the CSDL.
The proposed BenchFlow framework tackle R.G. 3 presented in Sect. 1.2.1 and contributes to the R.G. 4. It allows the scheduling of performance tests that are managed by the BenchFlow expert system, both from real users and from CICD processes (part of CSDL), relying on a CLI or the exposed RESTful APIs.

[12] https://swagger.io/, last visited on February 7, 2021
[13] https://swagger.io/tools/swagger-codegen/, last visited on February 7, 2021

Part III

Evaluation

Chapter 7

Evaluations

In this chapter, we present the evaluations we perform to validate our claims and the provided solutions. We present the expert review[1] we perform to collect expert feedback about the DSL and the overall proposed approach, and the summative evaluation we perform with IT practitioners participants where we propose tasks related to specifying tests using the implemented DSL for declarative performance testing. We then discuss the iterative reviews we perform while developing the overall approach, and in particular, the DSL and the BenchFlow framework, as well as we compare the approach with alternative approaches and solutions available in academic and industry literature.

7.1 Expert Review of the BenchFlow DSL

We ask targeted experts in the performance engineering domain, to provide feedback on the DSL and the overall proposed approach. The scope of the expert review is to assess the overall DSL model, expressiveness, effort and suitability, and approach. We also ask experts to compare the proposed approach with widely-used standard imperative approaches. In building the expert review we refer to the work by [Rodrigues et al., 2018] about a framework to evaluate the usability of DSL. Following the criteria in [Rodrigues et al., 2018], we define a heuristic evaluation, targeting potential expert users, and evaluating the *comprehension/learning* and *representatives* dimensions, among other assessed.

[1] https://www.usabilitybok.org/, last visited on February 7, 2021

7.1.1 Evaluation Questions and Hypotheses

The evaluation seeks to answer the following questions:

RQ1) The proposed DSL has good expressiveness to accommodate most of the requirements for automating performance test execution and integrating it in CSDL;

RQ2) The proposed approach has better usability, compared to standard imperative performance testing automation approaches;

RQ3) The proposed approach has less perceived effort, compared to standard imperative performance testing automation approaches;

RQ4) The proposed approach has better reusability, compared to standard imperative performance testing automation approaches;

RQ5) The proposed approach is well suited to support the definition and execution of performance tests for the target users;

RQ6) The proposed approach is better suited than standard imperative approaches to support the definition and execution of performance tests for the target users.

We also consider the following *null* (H_0) and alternative hypotheses (H_{alt}) for each of the above questions:

RQ1) H_0: The level of expressiveness perceived by the participants is considered medium-low. H_{alt}: The level of expressiveness perceived by the participants is considered high;

RQ2) H_0: The proposed approach has the same or worse usability when compared to standard imperative approaches. H_{alt}: The proposed approach has better usability when compared to standard imperative approaches;

RQ3) H_0: The proposed approach has the same or more perceived effort when compared to standard imperative approaches. H_{alt}: The proposed approach has less perceived effort when compared to standard imperative approaches;

RQ4) H_0: The proposed approach has the same or worse reusability when compared to standard imperative approaches. H_{alt}: The proposed approach has better reusability when compared to standard imperative approaches;

RQ5) H_0: The proposed approach is not or limited suited to support the definition and execution of performance tests for the target users. H_{alt}: The proposed approach is well suited to support the definition and execution of performance tests for the target users;

RQ6) H_0: The proposed approach is suited for the target users the same or less than standard imperative approaches. H_{alt}: The proposed approach is suited for the target users more than standard imperative approaches.

7.1.2 Evaluation Methods Overview

To collect feedback and answer to the research questions presented in Sect. 7.1.1, we propose an online anonymous survey to expert users in the performance engineering domain, both from academia and industry.

7.1.3 Data Collection Method and Survey Dissemination

We disseminate the survey using different communication mediums to target expert users. We rely on the following dissemination channels:

a) an email to the SPEC RG mailing list. The mailing list is received by all the member of the various SPEC RGs, experts in the performance engineering domain;

b) direct emails to all the people we collaborated with during the years in which this research has been developed;

c) direct emails to many different practitioners in various companies around the globe working on performance engineering in different domains.

We collected responses for 31 days, from July, 1st to July, 31st. We incentivized participants to respond, by proposing a raffle of three prices of the value of € 20 each, in the form of discount coupons for popular online websites. The raffle was to be performed only if reaching at least 15 responses. Since we obtained more than 15 responses, the raffle has been successfully performed on August, 15th and three winners have been awarded the prize. We limited the number of incentives and the total value, to stimulate participants to respond, but avoiding too big incentives producing side-effects in the quality of collected responses[2]. We also did not mention any details on the raffle at the beginning

[2] https://www2.deloitte.com/us/en/insights/topics/social-impact/the-craft-of-incentive-prize-design.html, last visited on February 7, 2021

of the survey, we only anticipated the possibility to participate in the raffle. All the details on the same have been provided at the end of the survey.
The expert review survey is available and left online at the following address: https://bit.ly/dpe-expert-review-survey. The survey is configured for accepting participant responses and allowing her to edit the response after submission in case it is needed. The overview of the received responses in not shown at the end of the survey, because some of the data contain sensitive information, for example, the email, and we can not filter the data to show due to limitations in the tool we rely on for building and administer the survey, i.e., Google Forms[3]. We constantly updated the total number of responses over the period in which we accepted responses so that participants were informed. We tracked the number of people opening the survey by relying on the bitly URL shortening service. 25 people in total opened the survey.

7.1.4 Structure of the Evaluation Survey

The evaluation survey we propose to participants for evaluation has the following structure:

1) an introduction section;

2) a section with questions meant to collect experience and expertise about the participants;

3) a section dedicated to present an overview of the proposed approach, as well as examples in using the DSL for declarative performance testing specification;

4) a section dedicated to questions related to the Research Questions (RQs) of the expert review;

5) a section dedicated to collect final additional feedback;

6) a final section, thanking the participants and asking them to optionally provide the email address to participate in the raffle, other than providing more details on the raffle.

In App. C we report the entire survey, and the list of questions and possible answers, while the complete survey can be consulted at https://bit.ly/dpe-expert-review-survey.

[3] https://docs.google.com/, last visited on February 7, 2021

7.1.5 Evaluation Procedure

Introduction

In the introduction section of the survey, we present the context and the scope of the work, the overall DPE proposal for automating performance test and integrating them in CSDL, the target systems and target users of the proposed approach, the objectives of the expert review as well as the target audience of the same. The target audience is identified as *performance engineering practitioners, both in academia and industry.* We then discuss the structure of the survey and provide an expected average completion time of *up to 35-45 minutes, of which up to 20-30 minutes for learning about the proposed approach, depending on participant's background* assessed with initial trials of the expert review. In this section, we also make sure to provide the users with all the information useful to complete the survey, for example, hints on how to navigate the survey, increase the font-size, and the availability of full-size images for all the examples part of the survey. These hints are necessarily due to limitations in Google Forms in these respects, and to make the participant comfortable in completing the survey.

Starter Questions

We ask participants about their experience and expertise. We evaluate the experience by requesting to indicate their last degree of education, their current professional role among a set of selected roles or by manually indicating it, the number of years of experience in the current professional role and the number of years of experience in the area of software performance testing, also as a side activity. We then request participants to report their main tasks and activities in their current role and we propose them a selection of concepts relevant in the domain of this work, requesting them to evaluate how familiar they are with those concepts on a 5-scale Likert scale [Likert, 1932] ranging from *Not at all* to *Expert*. The mentioned questions are all required to be complete in the survey.

Learning the DSL and the BenchFlow Approach

In this section of the survey, we present an overview of the proposed approach. We illustrate the YAML specification format the users of our approach are expected to master for declaratively specifying performance tests and configuring the test's automation process. We then present some examples of declarative

performance test specifications relying on the proposed YAML specification format.
We discuss what users are expected to state for declarative performance test automation, and in particular:

1) the test goal and its configuration, as well as the performance space to be explored if required by the goal (*Goal*, *exploration_space*);
2) the load function (*load_function*);
3) the performance metrics of interest (*observe*);
4) the criteria to determine if the test execution has to be terminated before its completion (*termination_criteria*);
5) the expected outcome of the test according to defined quality criteria (*quality_gates*);
6) the workloads used for the test and how to mix its operations (*workloads*, *Operations*, *mix*);
7) the system under test type, and its configuration (*sut*);
8) the data collection services needed to collect performance data useful to compute the metrics of interest (*data_collection*).

We then provide details for the CSDL extension of the DSL allowing users to specify test suites, focusing on:

1) the test suite, selecting tests by referring to YAML files defining them or by selecting them using labels (*suite*, *tests*);
2) the triggers activating the test execution, according to CSDL events (*triggers*);
3) the environments on which tests have to be executed (*environments*);
4) the expected outcome of the test suite according to defined quality criteria (*quality_gates*).

We also describe how the users are expected to define the tests and test suites bundle (including the SUT deployment descriptor), and how they, or a tool in CSDL, can submit them to the BenchFlow framework taking care of the

end-to-end automation in executing declarative performance tests as well as how they can access tests and test suites execution status and results. We also highlight test specifications that can be reused, extended, and overridden, to facilitate reuse of base test definitions for more complex tests.
We provide details for all the elements part of the YAML representation of the DSL for specifying tests and test suites, focusing on providing a good understanding of the different elements of the DSL and relevant details to enable users in understanding the overall vision of the approach and how she is expected to specify and execute tests, similarly to Sect. 5.6.1, and Sect. 5.6.2. After presenting the YAML specification, we provide in-depth and throughout examples on how to specify performance tests with different goals and test suites integrated in different moments of the CSDL, similarly to Sect. 5.6.3.

Expert Review Questions

We propose the expert with five 5-scale Likert questions, related to the evaluation questions and hypothesis to test presented in Sect. 7.1.1. For each of the five questions for which we require an answer, we also propose an optional question asking the participant to motivate the provided answer. To evaluate the *RQ1*, we ask the participants "How do you evaluate the overall expressiveness of the proposed DSL?". They can select among five possible answers ranging from *Very Poor* to *Excellent*.
To evaluate the *RQ2*, we ask the participants:

1) "How do you compare the perceived usability and effort of the proposed approach for performance test automation compared to the perceived usability and effort of standard imperative approaches?";

2) "How do you compare the perceived overall reusability of performance tests implemented using the proposed approach compared to the reusability of tests implemented using the standard imperative approaches?".

They can select among five possible answers ranging from *Very Poor* to *Excellent*.
To evaluate the *RQ1*, we ask the participants:

1) "How suitable do you consider the proposed approach for the target users of the same?". They can select among five possible answers ranging from *Not Suitable* to *Very Suitable*;

2) "How do you compare the suitability of the proposed approach compared to the suitability of standard imperative ones for the target users of

the proposed approach?". They can select among five possible answers ranging from *Much Worse* to *Much Better*.

Wrap-up Questions

Before concluding the survey we ask participants to report the main Pro's and Con's of the proposed approach according to them, as well as name similar approaches they are aware of.

7.1.6 Results

In this section, we discuss the results obtained from the performed expert review.

Number of Respondents

The total number of respondents of the survey is 18, compared to the number of people opening the survey page (25), it corresponds to 72.0%. The percentage is fairly high, very likely because of the way we disseminated the expert review, by targeting specific communities and contacts.

Respondent Experience and Expertise

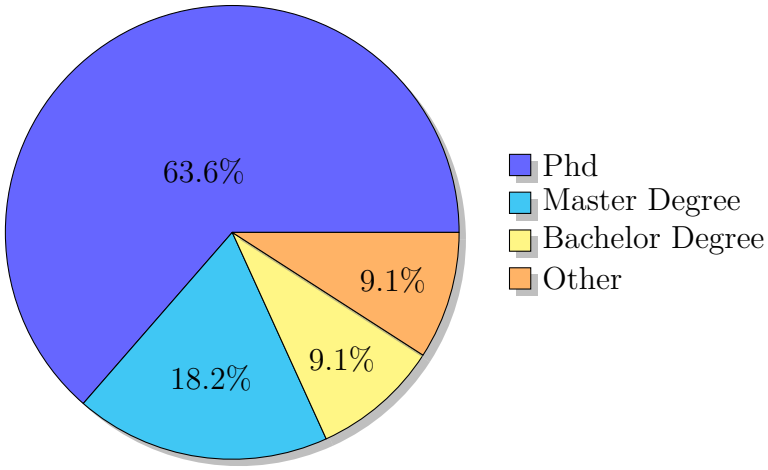

Figure 7.1. Expert Review: Education

Education - as reported in Fig. 7.1, most of the respondent have a PhD (63.6%). This is expected, given the target audience and the way we disseminate the survey. Another substantial part of the participants has a Master

Degree (18.2%) or a Bachelor Degree (9.1%). 9.1% of participants reported *Other* as the latest education level.

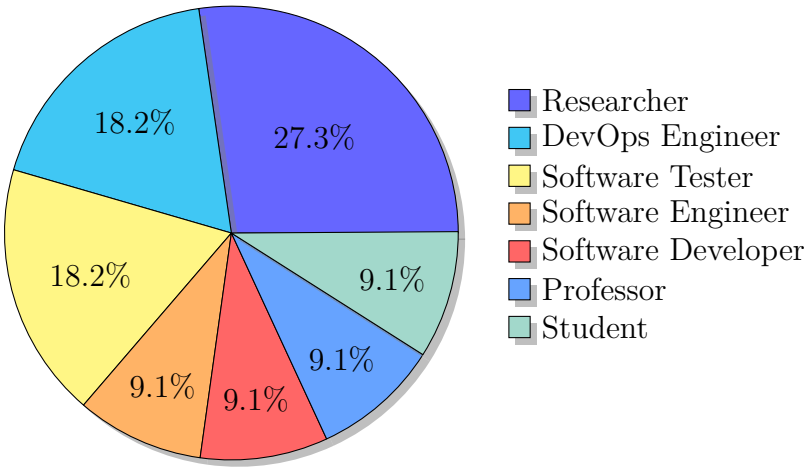

Figure 7.2. Expert Review: Professional Role

Professional Role - as reported in Fig. 7.2, most of the respondent are Researchers (27.3%), DevOps Engineer (18.2%) or Software Testers (18.2%). Other professional roles indicated by the participants are Software Engineer (9.1%), Software Developer (9.1%), Professor (9.1%), Student (9.1%).
Experience in the current professional role - the experience in the current professional role indicated by the participants is in the range of [3-10] with a median of 6 (reported by 36.4% of the participants). We can consider the participants with medium to high expertise in their role.
Experience related to software performance testing - the experience related to software performance testing indicated by the participants is in the range of [3-20] with a median of 13 (reported by 18.2% of the participants). The participant reporting three years of expertise is a Student. We can consider the participants with high expertise related to performance testing, thus valid target audience for our expert review.
Which are your main tasks and activities? - the main tasks and activities reported by the participants are related to implementing automation processes, defining performance tests, writing performance tests, and executing them, as well as raising software performance awareness. The activities of the participants are relevant to the scope of the expert review.
How familiar are you with the following concepts? - In Fig. 7.3 we present the assessment of the familiarity of participants with concepts we identified as relevant to properly understand and evaluate the proposed approach.

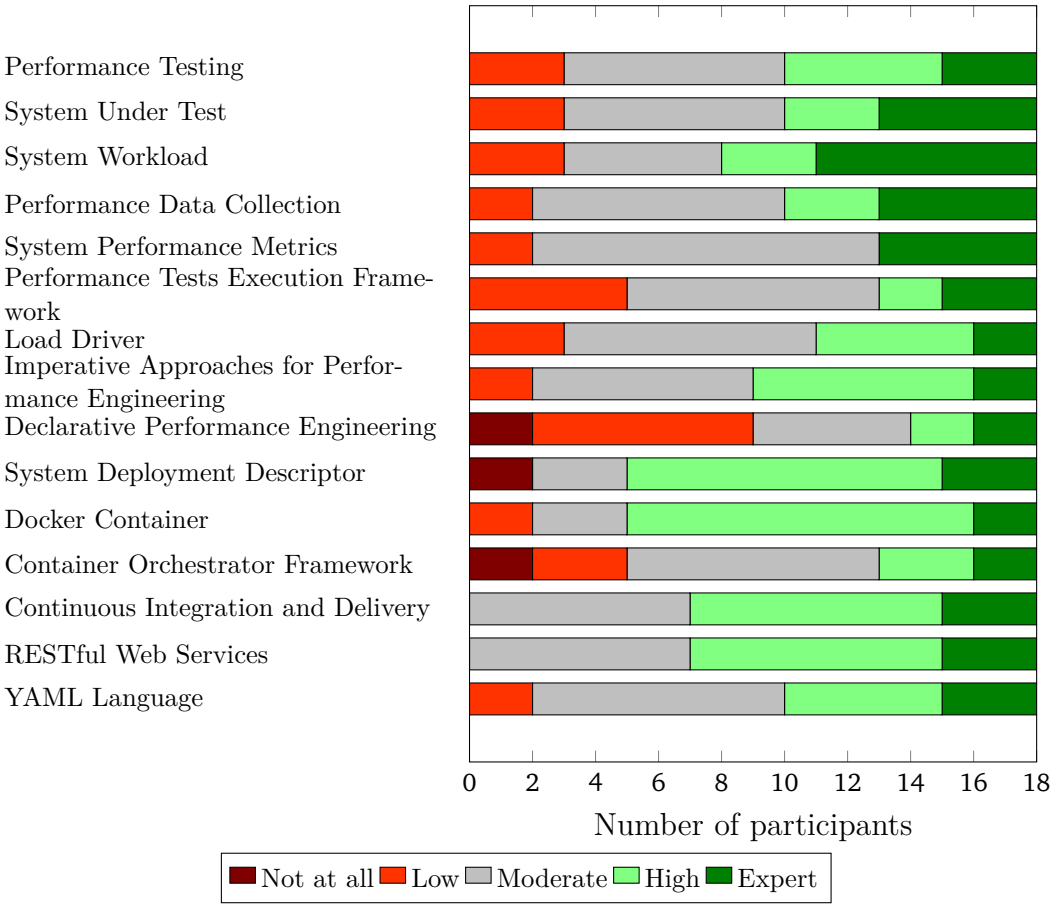

Figure 7.3. Expert Review: Familiarity with Concepts

We propose participants with concepts related to the performance testing domain, Docker domain, CSDL domain, RESTful Web service domain.
The respondent report to be:

1) *Moderate* to *Expert* (with a median on *High*) familiar with the performance testing domain;

2) *Moderate* to *High* (with a median on *Moderate*) familiar with the Docker domain;

3) *High* familiar with the CSDL domain;

4) *High* familiar with the RESTful Web service domain.

We also ask participants to indicate their familiarity with the DPE domain and the results indicate they have a *Low* to *Moderate* familiarity with such domain.

The collected feedback validates the participants are experts in the fields of interest for the proposed approach.

Expressiveness Results

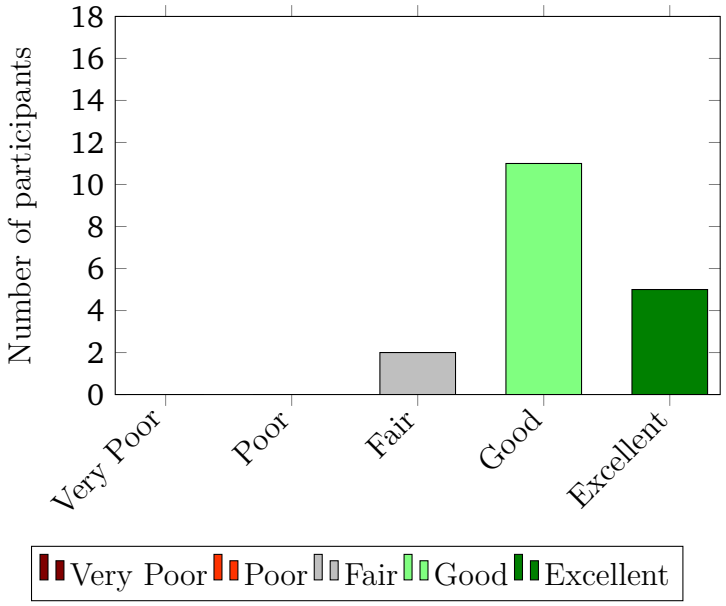

Figure 7.4. Expert Review: Expressiveness Evaluation

Expressiveness - In Fig. 7.4 we present the result on the expressiveness of the DSL. The participants evaluate the overall expressiveness of the approach in the range of [*Fair-Excellent*] with a median value on *Good*. 9.0% of participants indicate a *Fair* expressiveness, 64.0% of participants indicate a *Good* expressiveness, and the remaining 27.0% indicate an *Excellent* expressiveness. Most of the participants (81.0%) also provide additional motivations for their answers. The main constructive feedback is related to the complexity of expressing the workload and the need to extend it to other kinds of workloads, "[...]The workload specification offers quite some modeling mechanisms, but there are many more, e.g., using state machines or based on responses of the SUT (reacting differently to failed and successful logins)", and the fact YAML might be difficult to handle when specifying complex tests, "[...] The parameters of the YAML are overwhelming at first sight but necessary to configure a performance test". We also receive appreciation for the proposal, for example: "I think the language provides a very good coverage of the performance testing

domain it targets" and "It looks it can definitely be integrated with modern continuous delivery tools".

Usability and Effort Results

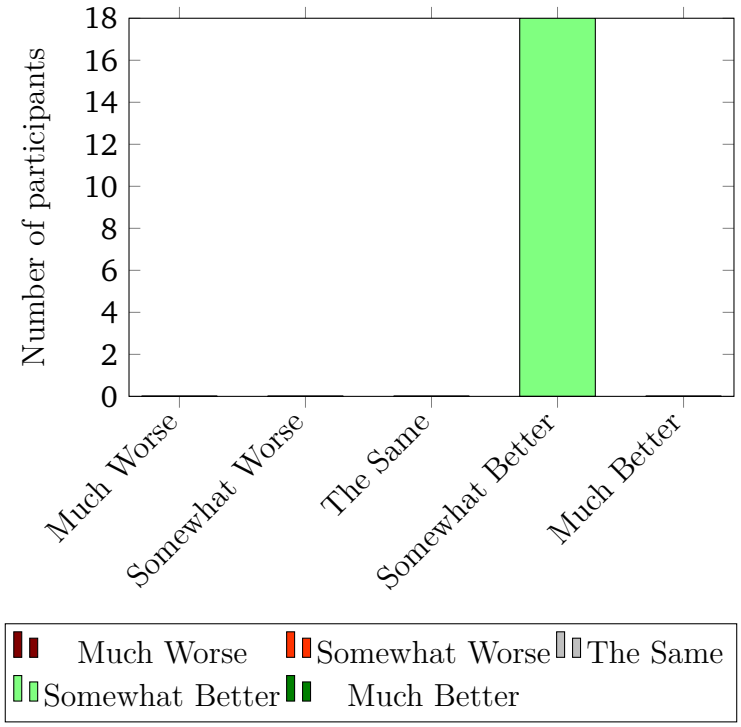

Figure 7.5. Expert Review: Usability Evaluation

Usability vs Imperative approaches - In Fig. 7.5 we present the result on the perceived usability of the DSL and the overall approach, compared to standard imperative approaches. The participants evaluate the overall expressiveness of the approach as *Somewhat Better* than imperative ones. 100.0% of participants indicate the perceived usability of the proposed approach as *Somewhat Better* than standard imperative approaches.
Most of the participants (72.0%) also provide additional motivations for their answers. The main constructive feedback is related to the need of providing support in editors: "I was easily able to understand the YAML specifications, but specifying them seems to require a lot of expertise in the DSL. For instance, some information is spread over different sections, which might make sense, but hinders straightforward specification. The support of a dedicated editor might

help[...]". One participant also reports: "Definitely novices benefit from the DSL".

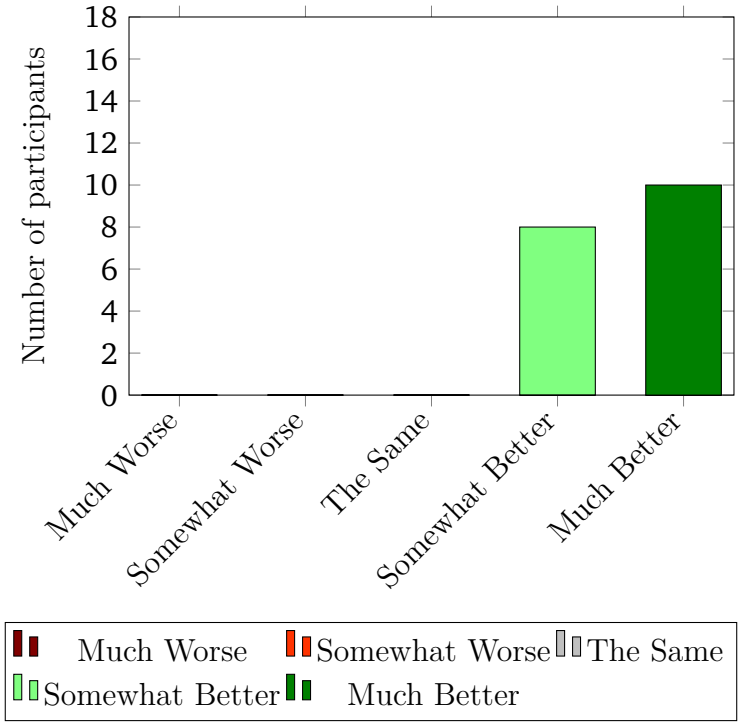

Figure 7.6. Expert Review: Effort Evaluation

Effort vs Imperative approaches - In Fig. 7.6 we present the result on the perceived effort of the DSL and the overall approach, compared to standard imperative approaches. The participants evaluate the overall expressiveness of the approach in the range of [*Somewhat Better-Much Better*] with a median value on *Much Better*. 45.0% of participants indicate the proposed approach is *Somewhat Better* in perceived effort compared to imperative approaches, and 55.0% of participants indicate the approach is *Much Better* in perceived effort compared to imperative approaches.
Most of the participants (72.0%) also provide additional motivations for their answers. One participant appreciates the BenchFlow framework enabling the automation, programmed using the DSL: "[...]The saved effort originates the framework that automatically deploys the SUT and executes the tests (potentially multiple times in a row), so I do not have to take care about it by myself".

Reusability Results

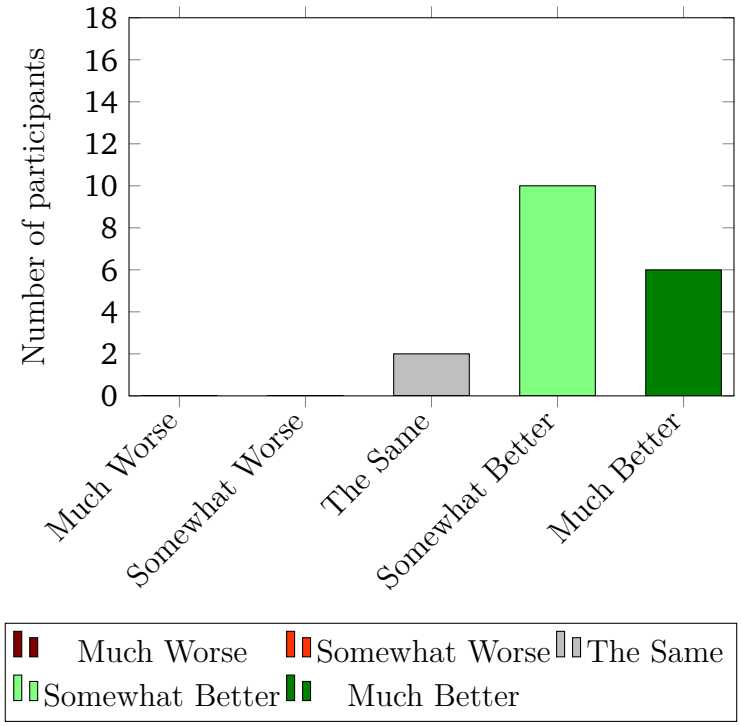

Figure 7.7. Expert Review: Reusability Evaluation

Reusability vs Imperative approaches - In Fig. 7.7 we present the result on the reusability of the DSL, compared to standard imperative approaches. The participants evaluate the overall reusability of the approach in the range of [*The Same-Much Better*] with a median value of *Somewhat Better* when compared to the reusability of standard imperative approaches. 9.0% of participants indicate the proposed approach is *The Same* in perceived overall reusability compared to imperative approaches, while 54.0% and 37.0% of participants indicate the approach is *Somewhat Better* and *Much Better* respectively in perceived effort compared to imperative approaches.
Most of the participants (72.0%) also provide additional motivations for their answers. The main feedback we received is: "Less modifications are needed. I guess it is easier also to apply versioning", "The declarative DSL eases reusing the same elements in multiple tests[...]", and "The proposed approach presents a fast way to configure a multitude of different performance scenarios in one go. In comparison to the standard approach, it is much more flexible and faster to execute for people with at least some knowledge in the field of performance

testing". One participant also reports: "I see I can cover many cases, and also share what I am building with people knowing less than me about performance testing".

Suitability for the Target Users Results

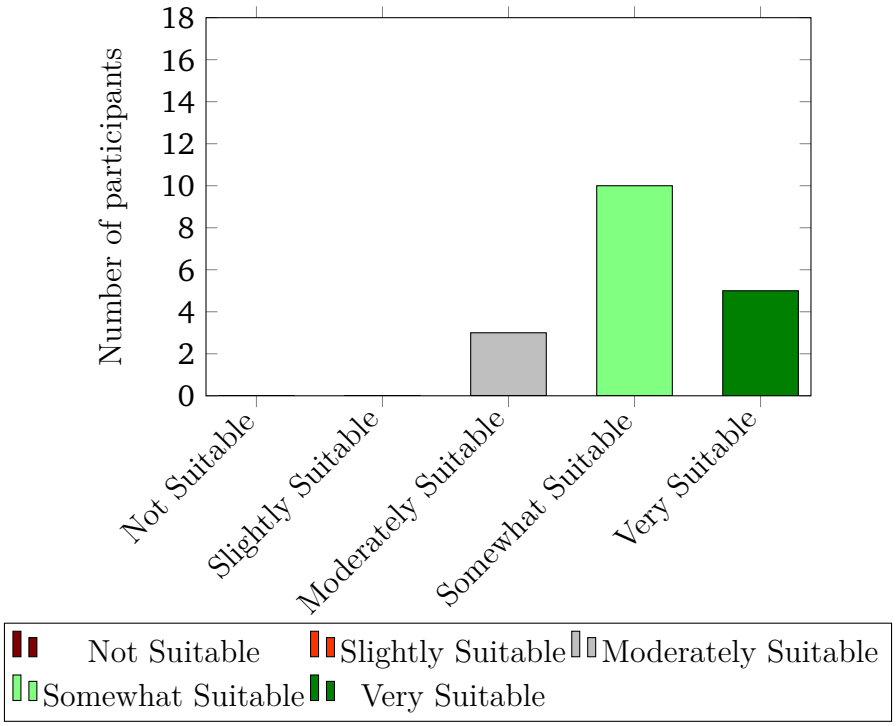

Figure 7.8. Expert Review: Suitability Evaluation

Suitability for the Target Users - In Fig. 7.8 we present the result on the suitability for the target users of the overall approach. The participants evaluate the overall expressiveness of the approach in the range of [*Moderately Suitable*-*Very Suitable*] with a median value on *Somewhat Suitable*. 18.0% of participants indicate the approach as *Moderately Suitable* for the target users, 54.0% of participants consider it *Somewhat Suitable*, and the remaining 28.0% indicate it is *Very Suitable*.

Suitability for the Target Users vs Imperative Approaches - In Fig. 7.9 we present the result on the suitability for the target users of the overall approach when compared to standard imperative approaches. The participants evaluate the overall suitability of the approach in the range of [*The Same*-*Much Better*] with a median value of *Somewhat Better* when compared to the

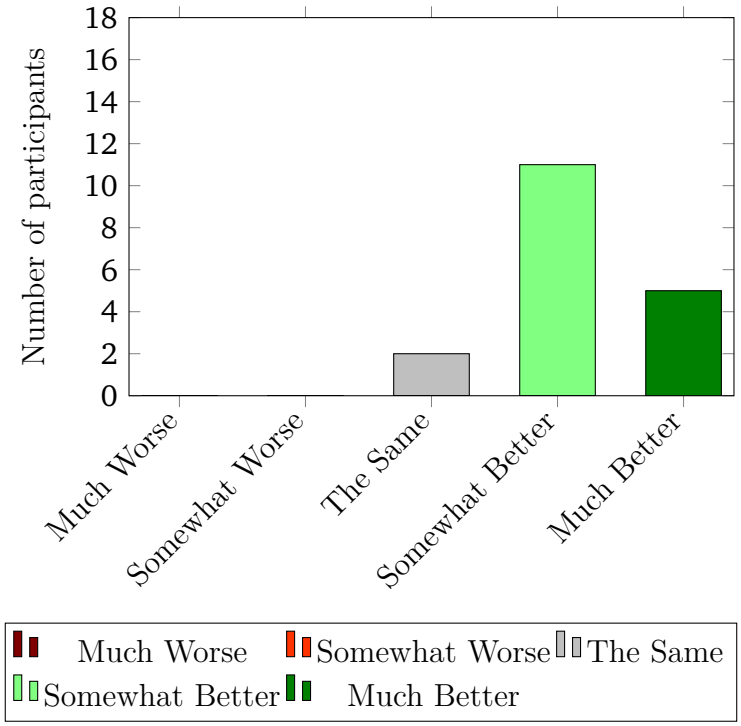

Figure 7.9. Expert Review: Suitability vs Imperative Approaches Evaluation

suitability of standard imperative approaches. 9.0% of participants indicate the proposed approach is *The Same* in overall suitability for the target users compared to imperative approaches, while 63.0% and 28.0% of participants indicate the approach is *Somewhat Better* and *Much Better* suitable respectively, for the target users, compared to imperative approaches.
Some of the participants (45.0%) also provide additional motivations for their answers. The main feedback is: "The DSL fits well to the YAML files used for Docker/Kubernetes/... and the integration with CICD is crucial in my opinion[...]", "A developer, QA engineer, or software tester is probably familiar with the domain of performance testing and can easily adapt to the proposed approach. This approach allows for a multitude of configurations in one file and does not necessarily rely on interfaces", "Definitely a step forward for performance testing awareness".

Wrap-up Considerations

We report the main Pro's, Con's, and similar approaches as reported by the participants.

Main Pro's - The main reported Pro's are:

1) "centralised configuration; simplicity";

2) "1. Automated deployment of the SUT. 2. Automated execution of multiple tests (and determination of the number of tests to execute). 3. Easily understandable DSL. 4. A single description of everything (workload, goal, quality gates, termination criteria, deployment, ...)";

3) "The YAML format provides a clear and structured document. Parameters in the configuration (i.e. resource) are easily visible. A configuration is reusable";

4) "The goal-driven specification, clearly stating the goal of the test. Usually, this is hindered in test documents".

Main Con's - The main reported Con's are:

1) "fine-tuning of the underlying tools (monitoring, load driver); still not human-readable, but machine-readable; more detailed workload specifications are not possible";

2) "[...]I did not see any support for specifying the workload, which can be very time consuming[...]";

3) "[...]Based on the performance scenario a configuration file can become very bloated and might be hard to follow[...]";

4) "As an expert I would need to learn something quite different compared to the tools I am using now";

5) "Time is required to get used to the specification, but I think anyway less than learning how to do it with imperative languages".

Similar Approaches - Reported similar approaches are:

1) "BDD/Gherkin, Declare";

2) "Taurus (gettaurus.org), artillery.io".

We include the reported similar approaches related to performance testing in the analysis we provide as part of the state-of-the-art analysis in Sect. 3.7.

Conclusion and Lessons Learned

The expert's feedback reported the expressiveness of the DSL is considered on median as *Good* in covering the needs of the target users and suitable for them in specifying and executing performance tests, and the reusability, the suitability of the approach for the target users, the perceived usability and effort is considered on median *Much Better* (the average, if existing, would lie in between *Somewhat Better* to *Much Better*) than standard approaches for performance testing, mostly imperative ones. In general, the approach we propose as an answer to our R.G. 2 and R.G. 3 is considered well suited in supporting target users of the same in definition and execution of performance tests. This allows to confirm all the alternative hypothesis we state in Sect. 7.1.1 for RQ1-6. The experts also provide us with valuable and constructive feedback we take into account for future work. Experts provide constructive feedback about the expressiveness of the DSL, hinting us to improve and facilitate the way users can specify the workload, constructive feedback about the usability, suggesting to us the need for supporting the users with an editor for test specifications writing.

7.2 Summative Evaluation of the BenchFlow DSL

As presented in Sect. 1.2, we aim at designing new methods and techniques for the declarative specification of performance tests suitable to be used alongside the CSDL. In doing so, as part of the R.G. 2, we developed a declarative DSL for the specification of performance test automation processes. We perform a summative evaluation[4] with potential target users of our approach, to assess the learnability of the proposed DSL (usability), and collect feedback on expressiveness, and reusability. The target users of our approach are practitioners involved in the CSDL, and in particular developers, software testers, quality assurance engineers, DevOps engineer, and operations engineers. In building the summative evaluation we refer to the work by [Rodrigues et al., 2018] about a framework to evaluate the usability of DSL. Following the criteria in [Rodrigues et al., 2018], we define a heuristic evaluation, targeting potential users of the proposed approach and evaluating the *comprehension/learning* and *representatives* dimensions, among other assessed.

[4] https://www.usabilitybok.org/, last visited on February 7, 2021

7.2.1 Evaluation Questions and Hypotheses

The evaluation seeks to answer the following questions:

RQ1) The proposed DSL has good learnability to support the target users in automating performance test execution and integrating it in CSDL;

RQ2) The proposed DSL has good reusability to support the target users in automating performance test execution and integrating it in CSDL.

We also consider the following *null* (H_0) and alternative hypotheses (H_{alt}) for each of the above questions:

RQ1) H_0: The level of learnability is medium-low. H_{alt}: The level of learnability is high;

RQ2) H_0: The level of reusability is medium-low. H_{alt}: The level of reusability is high;

We also propose other questions to the users, to collect general feedback on expressiveness, usability, the goal-based test specification, and comparison with widely-used standard imperative approaches.

7.2.2 Evaluation Methods Overview

To collect feedback and answer to the research questions presented in Sect. 7.2.1, we propose an online anonymous survey to potential target users, both from academia and industry.

7.2.3 Data Collection Method and Survey Dissemination

The data collection and survey dissemination methods for the summative evaluation are similar to the one we relied on for the expert review, presented in Sect. 7.1.3. Given the broader audience, we target for the summative evaluation, we rely on different channels and techniques compared to the expert review. This is important also to keep the target groups of the two evaluation as separate as possible. We disseminate the summative evaluation relying on:

a) direct emails to many colleagues and people we know in various companies around the globe working in one of the roles of the target user of our approach;

b) post on Linkedin[5], and Twitter[6] about the survey. The posts have been shared by many people to their respective contacts;

c) post in selected Reddit[7] communities focusing on: performance engineering, performance testing, performance testing related tools (e.g., JMeter), quality assurance, DevOps, Cloud-native technologies (e.g., Docker, Kubernetes), CICD. We also advertised the survey in larger communities in computer science, information technology, and programming, as well as in a community dedicated to posting surveys[8].

Reddit has been an excellent tool for disseminating the summative evaluation survey and collecting feedback [Shatz, 2017]. Almost a third of the views come from Reddit and about a quarter of the survey responses. We also engaged in interesting discussions with the targeted communities on Reddit, providing us with valuable and constructive feedback we consider when writing this dissertation.
The time-span in which we collected responses and the prize incentives are the same as the expert review. The minimum number of responses to be reached to perform the raffle is 40 for the summative evaluation, given we need more participants and the audience is broader. We reached the minimum number and we performed the raffle on August, 15th and three winners have been awarded the prize.
The summative evaluation survey is available and left online at the following address: `https://bit.ly/dpe-summative-evaluation-survey`. The settings of the survey are the same as per the expert review. We tracked the number of people opening the survey by relying on the bitly URL shortening service. 401 people in total opened the survey.

7.2.4 Structure of the Evaluation Survey

The evaluation survey we propose to participants for evaluation has the following structure:

1) an introduction section;

2) a section with questions meant to collect experience and expertise about the participants;

[5] `https://www.linkedin.com/`, last visited on February 7, 2021
[6] `https://www.twitter.com/`, last visited on February 7, 2021
[7] `https://www.reddit.com/`, last visited on February 7, 2021
[8] `https://www.reddit.com/r/SampleSize`, last visited on February 7, 2021

3) a section dedicated to present an overview of the proposed approach, as well as examples in using the DSL for declarative performance testing specification;

4) a section dedicated to multiple-choice tasks related to the RQs of the summative evaluation meant to assess the learnability and reusability of the proposed approach (mainly the DSL);

5) a section dedicated to questions about expressiveness, usability and other general questions about the goal-oriented specification and the overall approach when compared to standard imperative approaches;

6) a section dedicated to collect final additional feedback;

7) a final section, thanking the participants and asking her to optionally provide the email address to participate in the raffle, other than providing more details on the raffle.

In App. D we report the entire survey, and the entire list of questions and possible answers, while the complete survey can be consulted at `https://bit.ly/dpe-summative-evaluation-survey`.

7.2.5 Evaluation Procedure

Introduction

The introduction section for the summative evaluation is similar to the one on the expert review presented in Sect. 7.1.5. The target audience is identified as *developers, software testers, quality assurance engineers, and operations engineers, both in academia and industry.* The provided expected average completion time for the summative evaluation is *up to 35-50 minutes, of which up to 25-30 minutes for learning about the proposed approach, depending on your background.*

Starter Questions

The starter questions we propose participants of this survey are the same as the one proposed for the expert review and discussed in Sect. 7.1.5. The objective is to assess the participants' experience and expertise.

Learning the DSL and the BenchFlow Approach

The overview of the approach presented to the user to illustrate the proposed approach is similar to the one provided for the expert review and presented in Sect. 7.1.5. Given the different assessment method and the target to assess the learnability, we provide the users with fewer examples when compared to the expert review. We provide examples of how to specify a load test, a smoke test, and how to integrate smoke test suites as part of the CSDL. We make sure the examples do not collide with the assessment tasks we propose to the users.

Assessment Tasks

We propose the participants of the summative evaluation with different multiple-choice tasks covering all the elements of the DSL. We opt for a multiple-choice tasks survey to collect feedback about learnability and reusability because we rely on an online survey and more in-depth analysis, for example, an evaluation based on the actual writing of test specification, is not advisable with such a medium. The multiple-choice approach has been demonstrated a valid way to assess participants' ability to "recall information; [...] understand concepts; [...] solve problems" [Burton et al., 1990], thus a good approach to assess learnability and reusability. We rely on an online survey to scale with the collected feedback to a large number of participants. Due to the structure of the assessment, we do not include the standard System Usability Scale (SUS) for measuring usability. It would make limited sense due to the fact the users are not using a tool for completing tasks.
The tasks are divided into tasks assessing the learnability and tasks assessing the reusability. The tasks assessing the learnability of the proposed approach, related to *RQ1*, and in particular the DSL, cover:

1) the test goal (*goal*);

2) the load function (*load_function*);

3) the exploration space (*exploration_space*);

4) the performance metrics of interest (*observe*);

5) the criteria to determine if the test execution has to be terminated before its completion (*termination_criteria*);

6) the expected outcome of the test according to defined quality criteria (*quality_gates*);

7) the workloads used for the test (*workloads*);

8) the operations part of the workloads (*operations*);

9) the mix of operations (*mix*);

10) the system under test type, and its configuration (*sut*);

11) the data collection services (*data_collection*);

12) an entire declarative test specification (in the case of the example, a regression test);

13) an entire declarative test suite specification (in the case of the example, a test suite scheduling load tests).

The tasks assessing the reusability of the proposed approach, related to *RQ2*, and in particular the DSL, cover the reuse of:

1) an entire declarative test specification. In the case of the example the participants are requested to select a specification to reuse to specify an acceptance test;

2) a termination criteria specification;

3) a quality gates specification;

4) an entire test suite specification. In the case of the example, the reuse questions target the events to monitor for deciding when to schedule the test suite.

The tasks assessing the learnability report a DSL specification of a test or an entity part of the DSL and propose the users with three possible options about what the proposed specification represents, one of them being the correct answer. For all the questions related to learnability, we also request the participants to provide optional feedback about the showed specification. The tasks assessing the reusability report the user with three possible options to select from, representing DSL elements to be reused and a question asking them to select which of the three options they would select for accomplishing a proposed task. For both the tasks assessing learnability and the ones assessing reusability, the proposed three options are such that one is correct, one is

misleading on purpose, and a third one is wrong. We make sure the misleading answers across all the tasks have the same type of modification, according to the actual entity specification.
To reduce the possibility for the participants to identify patterns in the correct answer for the different tasks, we make sure to shuffle the possible options for each participant.

Post-Assessment Questions

After the tasks-based assessment, we ask the participants to reply to additional questions, using five 5-scale Likert, to collect general feedback on expressiveness, usability, the goal-based test specification, and comparison with standard imperative approaches. For each of the five questions, we also propose an optional question asking the participant to motivate the provided answer. The questions we propose are:

1) **Expressiveness** - "How do you evaluate the overall expressiveness of the proposed DSL?";

2) **Overall Usability** - "How useful do you consider the proposed approach for the target users of the same?";

3) **Goal-based specification Usability** - "How useful do you consider the goal-based specification for the target users of the same?";

4) **Comparison with standard imperative approaches** - "How much more/less do you think the proposed approach would help the target users of the same in implementing and executing performance tests, compared to the standard imperative approaches?".

Wrap-up

Before concluding the survey we ask participants whether they would be interested in using a tool based on the proposed approach, and to report the main Pro's and Con's of the proposed approach according to them, as well as name similar approaches they are aware of.

7.2.6 Results

In this section, we discuss the results obtained from the performed summative evaluation.

Number of Respondents

The total number of respondents of the survey is 63, compared to the number of people opening the survey page (401), it corresponds to 16.0%. The percentage is lower than the average reported in the literature (26.0%)[9], very likely because of the duration of the survey, requiring a considerable amount of time to be completed.

Respondent Experience and Expertise

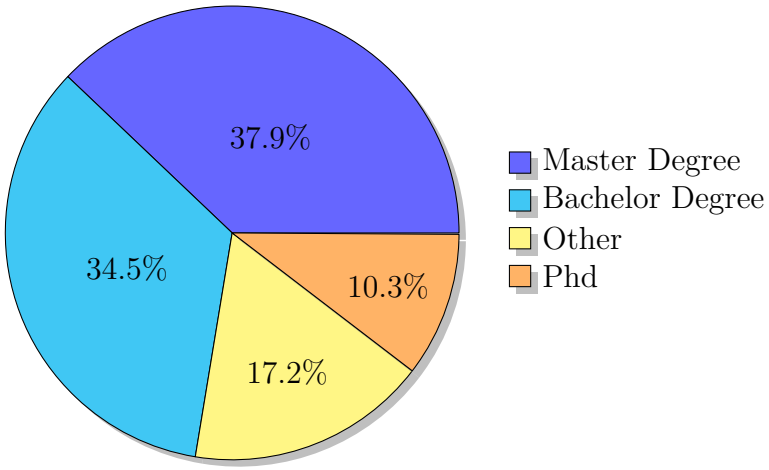

Figure 7.10. Summative Evaluation: Education

Education - as reported in Fig. 7.10, most of the respondent have either a Master Degree (37.9%) or a Bachelor Degree (34.5%). 17.2% of participants reported *Other* as the latest education level, while 10.4% of participants have a PhD.
Professional Role - as reported in Fig. 7.11, most of the respondent are Software Engineer (27.6%), DevOps Engineer (17.2%) or Software Developers (17.2%). Other professional roles indicated by the participants are Software Tester (10.3%), Researcher (6.9%), Site Reliability Engineer (6.9%), Student (6.9%), Performance Analyst (3.4%), QA Analyst (3.4%). The participants' professional role in line with the target users, with a bias towards roles traditionally less related to performance testing activities.
Experience in the current professional role - the experience in the current professional role indicated by the participants is in the range of [1-12] with a

[9] https://www.getfeedback.com/resources/online-surveys/better-online-survey-response-rates/, last visited on February 7, 2021

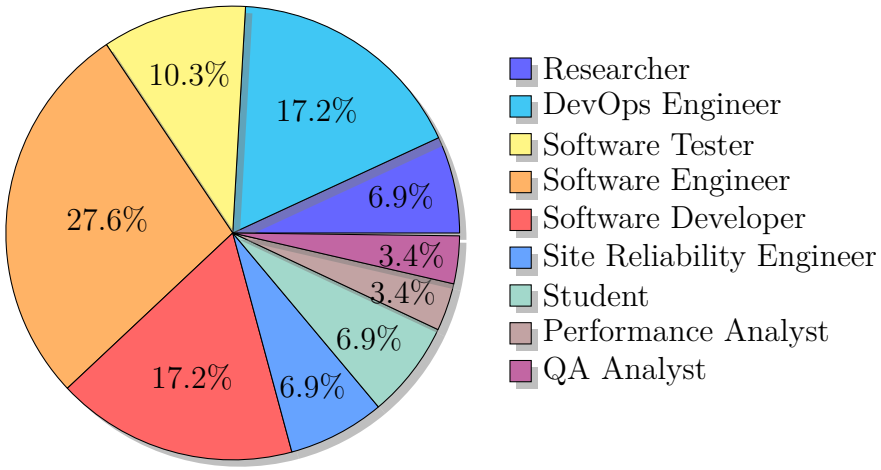

Figure 7.11. Summative Evaluation: Professional Role

median of 5 (reported by 17.2% of the participants). We can consider the participants covering expertise in their role from novice to expert.

Experience related to software performance testing - the experience related to software performance testing indicated by the participants is in the range of [0-6] with a median of 1 (reported by 34.5% of the participants). We can consider the participants with limited expertise related to performance testing, thus valid target audience for our summative evaluation.

Which are your main tasks and activities? - the main tasks and activities reported by the participants are related to architecting and developing software in different domains, research in different areas of computer science, automation related to CICD and running software.

How familiar are you with the following concepts? - In Fig. 7.12 we present the assessment of the familiarity of summative evaluation participants with concepts we identify as relevant to properly understand and evaluate the proposed approach. The assessed concepts are the same as for the expert review. We propose participants with concepts related to the performance testing domain, Docker domain, CSDL domain, RESTful Web service domain. The respondents report to be:

1) *Low* to *Moderate* (with a median on *Moderate*) familiar with the performance testing domain;

2) *Moderate* to *High* (with a median on *Moderate*) familiar with the Docker domain;

3) *High* familiar with the CSDL domain;

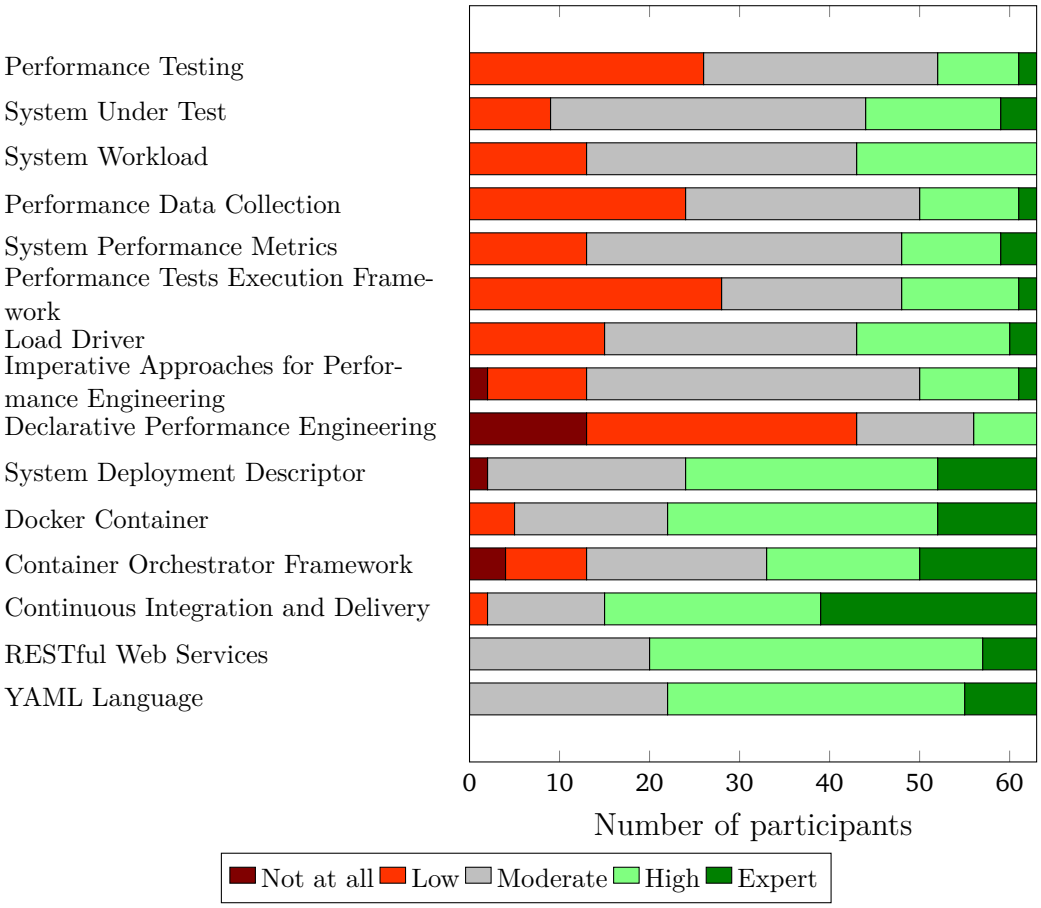

Figure 7.12. Summative Evaluation: Familiarity with Concepts

4) *High* familiar with the RESTful Web service domain.

We also ask participants to indicate their familiarity with the DPE domain and the results indicate they have a *Low* to *Not at All* familiarity with such domain. The collected feedback validates the participants have some understanding and knowledge about the performance domain and have a good to high understanding of concepts related to DevOps, CSDL, and CICD. They can be considered a good sample of target users.

Task-based Evaluation Results

Learnability Assessment Results In Fig. 7.13 we report the results of the tasks dedicated to assessing the learnability of the DSL and the overall proposed approach. Most of the participants correctly answer all the proposed questions (82.0% of the total number of participants), the remaining 18.0% wrongly

answer to at least one question, and in most cases by selecting the misleading answer. The task with the most correct answers (89.7%) is the one about the workload specification, while the tasks with the most, not correct answers are the one related to termination criteria (20.7%, of which 17.3% selected the misleading answer), SUT specification (20.7%, of which 17.3% selected the misleading answer), and complete test specification about the regression test (20.6%, of which 17.2% selected the misleading answer). We consider the task about the termination criteria specification as complex, and we expected it to be among the ones most difficult to learn. The same holds for the task on the regression test specification. We did not expect the task about the SUT specification to be difficult to learn. This result is probably due to *Moderate* familiarity with concepts related to Docker. The other tasks have a percentage of correct answers in the range [82.8%-86.9%].

To account for possible impacts on the results from participants identifying themselves as experts, independently on the number of years of expertise, on the concepts related to the performance engineering domain, in the assessment presented in Sect. 7.2.6, we compute the statistics related to the learnability of the task also by excluding them. In detail, we exclude from the set of the participants all the participants indicating a *High* to *Expert* familiarity with at least half of the performance domain concepts. This resulted in an updated set of 52 participants, effectively reducing the number of participants to about 17.0%. Almost all the remaining participants have either a Master Degree or a Bachelor Degree, they work as Software Engineer, DevOps Engineer, Software Developers, or Student. They work in their current role with a median of 6 years (with a range in [1-12]) and have on median 0 years of experience related to the software performance testing domain (with a range of [0-4]). They report a *Low* to *Moderate* (with a median on *Low*) familiarity with the performance testing domain. The results assessed on the updated set of participants do not deviate much from the one including participants more familiar with the performance engineering domain, although they demonstrate a slight reduction in correct answers as expected. Most of the participants from the updated set correctly answer all the proposed questions (76.0% of the total number of participants), the remaining 24.0% wrongly answer to at least one question, and in most cases by selecting the misleading answer.

Participants also contribute with feedback while performing the tasks, although in a much-limited percentage when compared to the expert review. We receive feedback from only 2 participants and only on the test suite specification. The participants report: “I like it! It looks very simple to integrate performance test suites with the tools I use every day to continuously build and deploy the

software I develop" and "I want this for my Jenkins pipelines :)".
Results on learnability can be considered very good. Although with limited time and information about the proposed DSL and overall approach, more than 80.0% (or almost 80.0% for the updated set of participants) correctly answer the proposed questions.

Reusability Assessment Results In Fig. 7.14 we present the results of the tasks we ask the participants to complete to assess the reusability of the DSL in the context of the proposed approach. The test suite and the termination criteria specification result in the ones with the most correct answers (82.8%). The goal specification reuse obtains 79.3% correct answers, while the termination criteria specification reuse obtains 65.5% correct answers and the most selection of the misleading answers (20.7%). The result of the reusability of the termination criteria is expected, given the result on the learnability of the same concept.
We assess the reusability on the same updated set of participants identified for the learnability assessment. Also in this case the results are in line with the ones considering all the participants, although they demonstrate a slight reduction in correct answers as expected. Most of the participants from the updated set correctly answer all the proposed questions (77.3% of the total number of participants), the remaining 22.7% wrongly answer to at least one question, and in most cases by selecting the misleading answer.
Results on reusability are in line with the one on learnability and can be considered very good. More than 80.0% (or almost 80.0% for the updated set of participants) correctly answer the proposed questions.

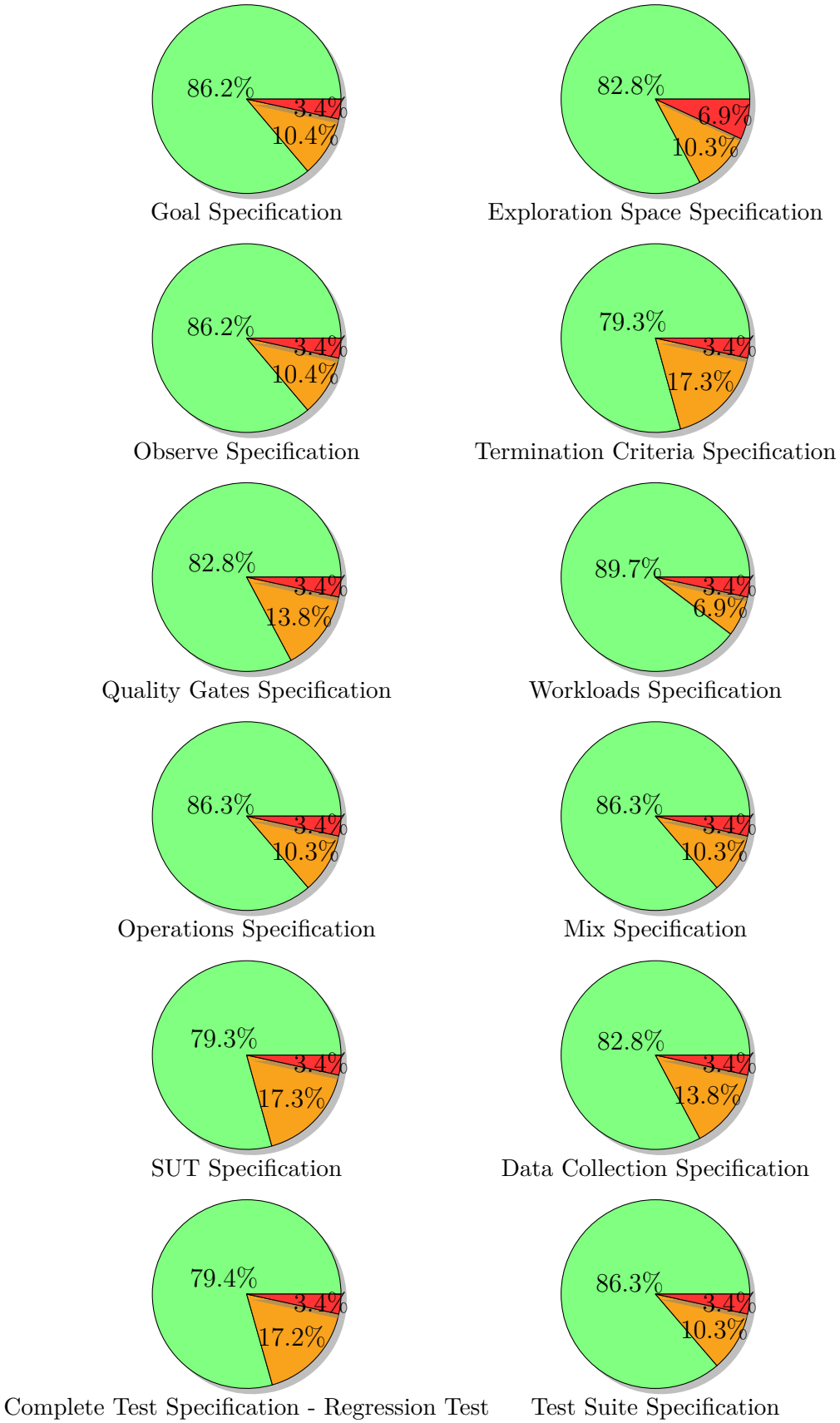

Figure 7.13. Summative Evaluation: Learnability Evaluation

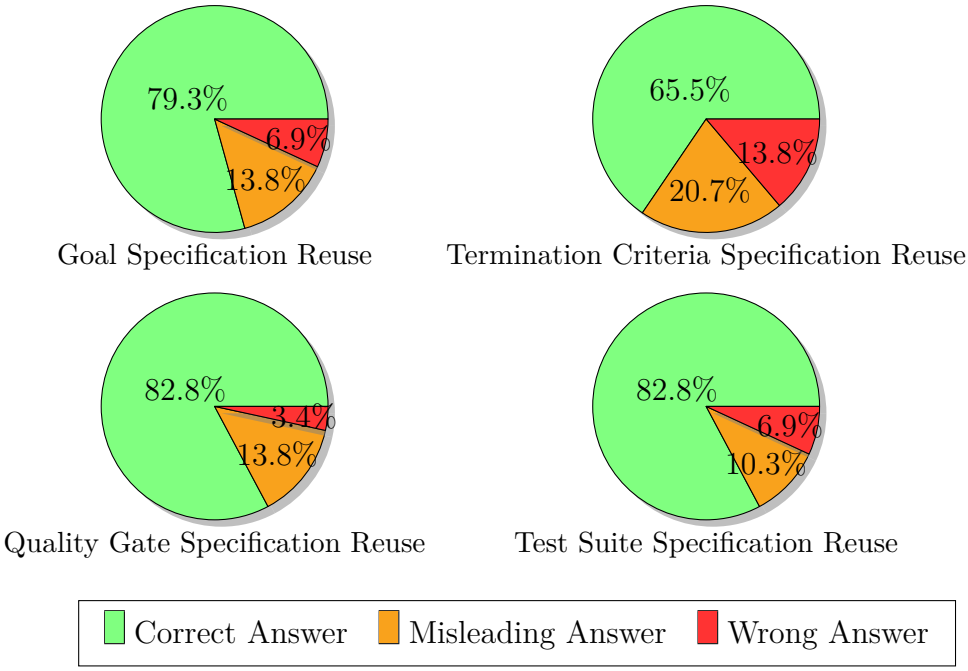

Figure 7.14. Summative Evaluation: Reusability Evaluation

Post-Assessment Considerations

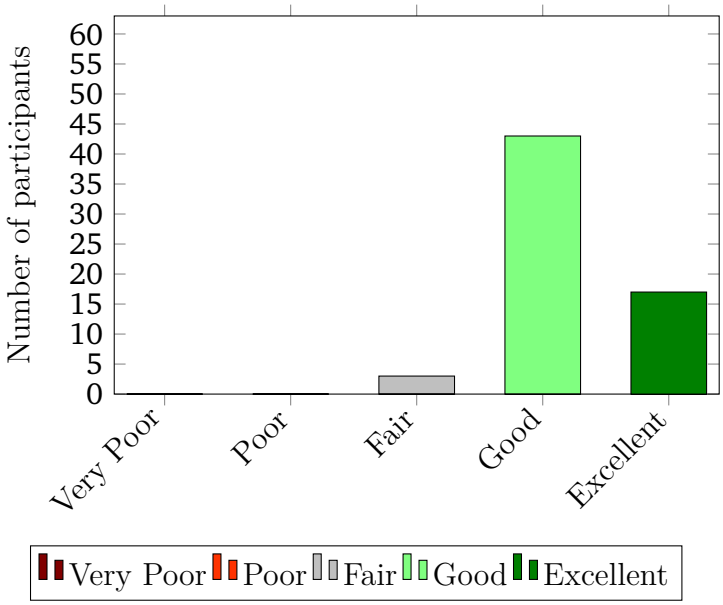

Figure 7.15. Summative Evaluation: Expressiveness Evaluation

Expressiveness In Fig. 7.15 we report the evaluation the participants of the summative evaluation give to the expressiveness of the proposed DSL. The participants evaluate the overall expressiveness of the approach in the range of [*Fair-Excellent*] with a median value on *Good*. 3.4% of participants indicate a *Fair* expressiveness, 69.0% of participants indicate a *Good* expressiveness, and the remaining 27.6% indicate an *Excellent* expressiveness.
Few participants (10.3%) also provide additional motivations for their answers, for example: "I like the fact I understand what I am requesting to be tested, even if I have almost no knowledge in performance testing" and "I find most of what I expect. Workload could integrate external definitions". Also from the summative evaluation, we receive feedback about the workload, and the opportunity to extend the workload definition semantics.

Overall Usability In Fig. 7.16 we present the assessment on the usability of the proposed approach for the target users, the participants provided. The participants evaluate the overall usability of the approach in the range of [*Moderately Usable-Very Usable*] with a median value of *Somewhat Usable* when referring to the target users of the approach. 6.9% of participants indicate the proposed approach is *Moderately Usable* for the target users, while 62.0%

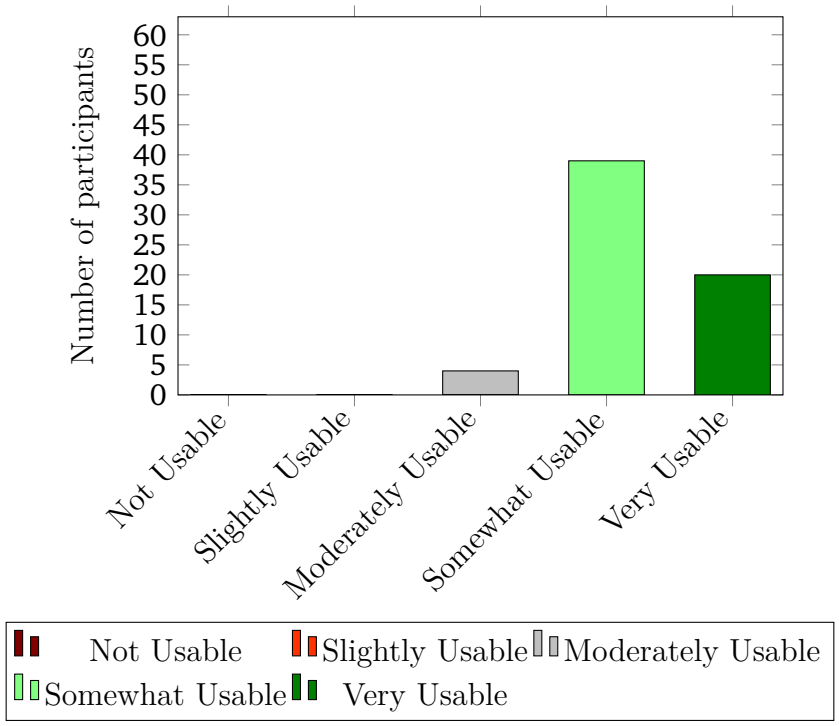

Figure 7.16. Summative Evaluation: Usability Evaluation

and 31.1% of participants indicate the approach is *Somewhat Usable* and *Very Usable* respectively.
Few participants (10.3%) also provide additional motivations for their answers, for example: "I think most of my colleagues would understand most of it" and "It looks like starting from template I can achieve many tests".

Goal-based Specification Usability In Fig. 7.17 we present the assessment on the usability of the goal-based declarative specification we propose as part of the DSL. The participants evaluate the overall usability of the goal-based specification in the range of [*Moderately Usable*-*Very Usable*] with a median value of *Somewhat Usable* when referring to the target users of the approach. 3.4% of participants indicate the proposed approach is *Moderately Usable* for the target users, while 51.7% and 44.8% of participants indicate the approach is *Somewhat Usable* and *Very Usable* respectively.
Few participants (10.3%) also provide additional motivations for their answers, for example: "I get what the test is defining, and that is great!" and "It is nice to see the goal clearly stated in the test".

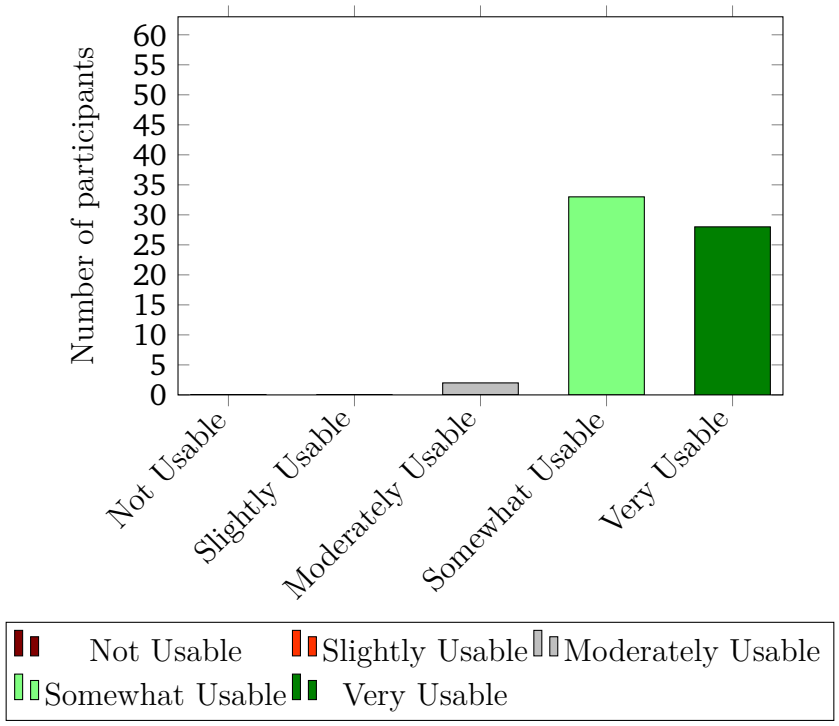

Figure 7.17. Summative Evaluation: Goal-based Specification Usability Evaluation

Comparison with Standard Imperative Approaches In Fig. 7.18 we illustrate the result about what participants think of the proposed approach compared to standard imperative approaches. We request participants to indicate how much more or less they think the proposed approach would help the target users of the same in implementing and executing performance tests, compared to the standard imperative approaches. The participants evaluate the approach in the range of [*Somewhat Less-Much More*] with a median value of *Somewhat More*, in helping target users in implementing and executing performance tests when compared to standard imperative approaches. 3.4% and 3.4% of participants indicate the proposed approach is *Somewhat Less* or *The Same* effective, respectively, in helping the target users compared to imperative approaches, while 48.3% and 44.8% of participants indicate the approach as *Somewhat More* and *Much More* effective, respectively, in supporting the target users in defining and executing performance tests.
Few participants (10.3%) also provide additional motivations for their answers, for example: “This is only because I think people are resistant to change” and “It is the way to go”. In particular, the feedback related to the resistance to change has been provided from the set of participants indicating the proposed

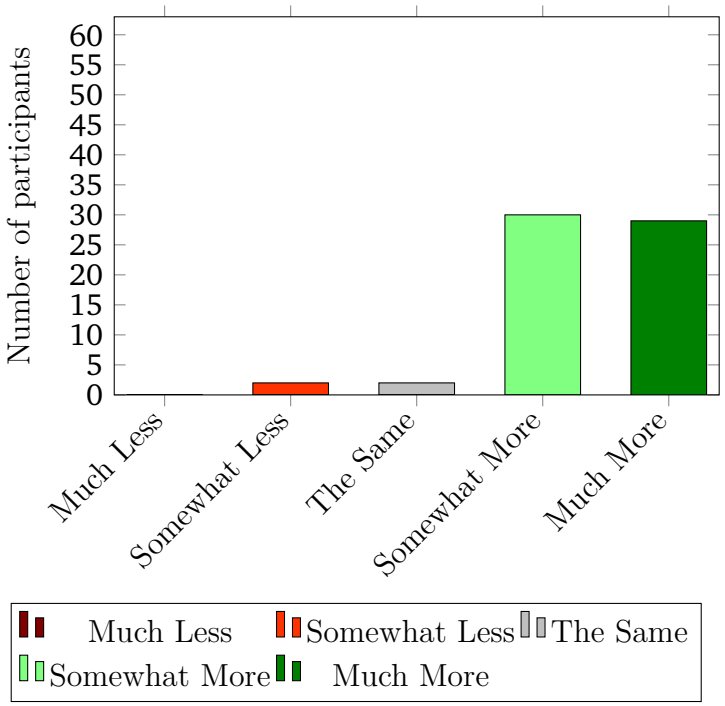

Figure 7.18. Summative Evaluation: Suitability vs Imperative Approaches Evaluation

approach is *Somewhat Less* effective in supporting the target users.

Wrap-up Considerations

We report whether participants are willing to use our approach, the main Pro's, Con's, and similar approaches as reported by the participants. None of the previous questions are mandatory.
When requested if they would be interested in using the framework implementing the proposed approach, 78.6% of the participants report they want to use it, while the remaining 21.6% report they are *Maybe* interested in using it. A total of 89.0% of the participants reply to the previous question.
Main Pro's - The main reported Pro's, reported by 20.7% of the participants, are:

1) "Easy to understand and guides you to a correct performance test definition";

2) "Easy to use DSL, users can read it and reasonably guess what's expected of the tests";

3) "it looks very simple and powerful. If it works as it is defined, I want to use it";

4) "I think some of the goals are very useful to test the performance of some embedded systems I develop".

The received feedback about the pro's of the proposed approach is in line with the one received as part of the expert review.
Main Con's - The reported Con's, received by a single participant, is:

1) "YAML is ugly, deep nesting can get confusing sometimes. Maybe abstracting away parts of the specification one can generate with a tool would help".

This is in line with feedback received about the complexity of the specification, and the need for IDE support we receive from experts participating in the expert review. The participant also adds the opportunity to introduce generated declarative test specifications, in line with the overall vision of DPE.
Similar Approaches - No similar approaches are mentioned by the participants.

Conclusion and Lessons Learned

The summative evaluation results indicate the learnability and reusability of the DSL is medium-high, considering more than 80.0% of participants on average correctly answer to the assessment tasks. In general, the approach is considered highly expressive in allowing users specifying performance tests and their automation process. When requested about the overall usability of the approach, the participants consider it to be between *Somewhat Usable* and *Very Usable*, including for what concerns the goal-based specification. The participants also assess the approach as better suited to effectively supporting the target users in defining and executing performance tests, when compared to standard imperative approaches. This allows to confirm all the alternative hypothesis we state in Sect. 7.2.1 for RQ1-2. The participants also express their willingness to use the framework implementing the proposed approach we developed as part of the work contributing to answering R.G. 3, for their performance testing activities. This validates the importance of providing a framework for implementing the overall proposed approach. The participants also provide constructive feedback about the need for tool support during the test specification writing and the opportunity to generate part of the specification instead of requesting the user to write all of it. We take into account the provided feedback for future work.

7.3 Iterative Review of the BenchFlow Framework

BenchFlow evolved over many years of use and iterative development. We initially developed BenchFlow for BPMN 2.0 WfMSs benchmarking, and then we evolved it to support RESTful Web Services and different case studies developed with different students and research groups around the world. Since the first version BenchFlow always embraced a declarative approach for performance testing.

	Features Overview
Before V1	*Features* • Only the experiment model is available and consequently, the goal-driven specification is not available. • The experiment model ad the framework focuses mainly on automating BPMN 2.0 WfMSs benchmarking. • To execute multiple experiments configuring the system in different ways requires the definition of multiple experiment specifications. • Quality gates are not available, and only time-based termination criteria are implemented. • Only Faban is available as a trial execution framework. *Rational* The proposed approach mainly focused on BPMN 2.0 WfMSs benchmarking.

(To be continued)

	Features Overview
V1	*Features* • The test model is added. • Support for exploration testing goal is added. • The exploration space configuration only supports the exploration of the number of users. • The only supported exploration strategy is the ”one-at-a-time” one. • Termination criteria based on the precision and the quality of collected data are supported. • The experiment model evolves in multiple iterations to support the features introduced in the test model. *Rational* Complex automation processes needed for automating BPMN 2.0 WfMSs benchmarking, testing the performance under different users load. Collaborating with other researchers in the BPMN 2.0 WfMSs benchmarking domain, we realized the importance of more advanced termination criteria.

(To be continued)

	Features Overview
V1.1	*Features* • Quality gates are supported. • More data collectors and metrics are computed. • More descriptive statistics and statistical tests are supported. • More performance test goals are supported. • The experiment model evolves in multiple iterations to support the features introduced in the test model. *Rational* Execution of multiple benchmarks on multiple BPMN 2.0 WfMSs, in different versions and configurations, required automatic validation of results and reporting. Collaborating with other researchers in the BPMN 2.0 WfMSs benchmarking domain, we realized the importance of supporting more descriptive statistics and statistical tests.

(To be continued)

	Features Overview
V2	*Features* • Web services performance testing is supported, adding the possibility to define complex $HTTP$ workloads. • JMeter is supported as a trial execution framework. • All the performance test goals are supported. • The exploration space configuration supports also the exploration of services. • The stability test and stability criteria are supported. • SUT deployment is not enforced anymore, and an endpoint of an already deployed SUT can be provided. • The *light* version of the BenchFlow framework is introduced, mainly to avoid deployment of the *analysis* related services. • The experiment model evolves in multiple iterations to support the features introduced in the test model. *Rational* Support Web services performance testing and more complex workloads and performance test goals required to properly assess the performance of complex Web services deployments (e.g., Microservices). In this iteration, many different elements of the DSL are redesigned and refactored to support the addition of Web services performance testing while collaborating and collecting feedback from many research groups working on Web services automation and declarative performance engineering. Collaborating with other research groups we also realized in some contexts SUT deployment facility and metrics computation is not required and is considered overhead.

(To be continued)

	Features Overview
V3	*Features* • The exploration strategy supports different approaches to explore the performance space. • Predictive exploration strategies (e.g., MARS) are supported. • Abstraction and integration with the CSDL are supported. • The experiment model evolves in multiple iterations to support the features introduced in the test model. *Rational* Optimize the execution of performance tests and enable the integration with CSDL to enable automation and integration with enterprise tools. In this iteration, we collaborated with companies interested in exploring declarative performance testing and we collected feedback on further improving the DSL and the BenchFlow framework.

Table 7.1. Development Iterations of the BenchFlow Framework

In Tab. 7.1 we present the main iterations of the BenchFlow development and the rationale for the different evolution phases as well as the feedback we collected motivating the evolution and the context in which such feedback has been collected.
Before V1 - the iterations before the V1 are mainly focused on supporting BPMN 2.0 WfMSs benchmarking. The test model is not part of the proposed approach yet, and to execute multiple experiments many different experiment specifications have to be provided. Faban is the only trial execution framework supported, and only time-based termination criteria are available to set an upper bound in the total execution time for an experiment. **V1** - we start to collaborate with other researchers interested in using the proposed approach

to automate their experiments. In V1 the test model is added, and a more complex performance test and performance automation process configuration are supported. In this version, relying on the feedback of researchers we collaborated with, we introduce termination criteria based on the quality of results. To support the test model and the new features of the approach, the experiment model evolves as well. This is the first version contributing towards the declarative performance specification of performance test, thus to the R.G. 2 presented in Sect. 1.2.1. **V1.1** - more advanced control on the performance test automation process is enabled, by introducing the quality gates. Benchmarking BPMN 2.0 WfMSs and supporting other researchers in their performance testing automation needs, we identified the need of adding more advanced descriptive statistics and statistical tests to ensure and validate the quality of collected performance data and computed performance metrics. V1.1 also supports more performance test goals, to accommodate for the different needs in systems' performance testing. **V2** - in V2 we add the support for performance testing of Web services, as part of a collaboration with other researchers in extending the BenchFlow framework and the DSL for Microservices performance assessment. Many DSL elements are redesigned for enabling the extension to Web services. To support Web services performance testing we also add support for JMeter as a trial execution framework. JMeter enables more complex workload specifications when compared to Faban. V2 enables all the performance test goals presented in Fig. 5.17, and allows for the execution of performance tests while skipping the SUT deployment part, when the SUT is deployed by third-party solutions or manually. In V2 we also introduce a *light* version of the framework. Working with other research groups we realized in some contexts the data analysis capabilities are not required, for example when collecting performance data that are then fed into performance modeling algorithms. Given the analysis component is quite complex to deploy and computationally intensive, in the light version we support the possibility of relying on the BenchFlow approach for declarative performance test automation and data collection only. The V2 is the first complete version contributing towards declarative and automated execution of performance tests, thus contributing to the R.G. 3 presented in Sect. 1.2.1. **V3** - completes the definition on the DSL features and the BenchFlow framework model-driven capabilities. We introduce the support for predictive performance space exploration (e.g., the MARS model) and different approaches for exploring the performance space. Collaborating with companies exploring the proposed DPE approach for performance testing automation, we also extend the DSL to support for declarative specification of test suites enabling the integration of declarative performance

testing with CSDL, as part of the R.G. 4 presented in Sect. 1.2.1.
The BenchFlow framework has been developed with the contribution of many students and researchers, improving its quality and extending its features. Over the four years from 2014 to 2018, five students contributed to the BenchFlow framework with the work they did as part of their Master theses.
The BenchFlow framework has been tested in the field collaborating with many research groups, research initiatives, and some companies. The collaboration enabled us to iteratively improve and perfect the framework and the overall approach towards the presented declarative specification of performance test automation processes. We collaborated with three main research groups, located in Italy, Germany, and across the United States of America and Brazil. We also collaborated with researchers mainly located in Germany, Italy, Switzerland, France, and Brazil. We deployed the BenchFlow framework in the three main research groups we collaborated with, and on the Public Cloud on Microsoft Azure[10] collaborating with researchers in France. All-in-all, for about four years from 2014 to 2018,

a) seven people actively contributed code to the BenchFlow repository on GitHub;

b) three main research groups adopted the proposed approach for their performance experiments automation needs;

c) more than 400 unique performance test bundles and more than 40 test suite bundles have been defined relying on the proposed DSL, covering all the test goals the DSL support with about 40.0% of them defining load tests. All the test executions that happened from V1.1 on, have been validated with quality gates. The different defined test bundles target both the SUT types we consider as target system and cover 20 different systems and 64 different versions in total;

d) more than five thousand performance experiments have been executed by the BenchFlow framework, starting from the defined bundles and accounting for the number of repetitions;

e) validation of performance tests, termination criteria, and failure handling mechanism in place as part of the BenchFlow framework avoided the execution of about 600 experiments. This number does not include termination criteria based on time and stopping the test execution, but only termination criteria based on the quality of results;

[10] https://azure.microsoft.com/, last visited on February 7, 2021

f) more than one terabyte of zipped performance data have been collected across the executed performance tests and has been analyzed with the BenchFlow framework;

g) more than 30000 metrics, statistics, and statistical tests have been computed to assess the quality of the SUTs target of the thousands of executed performance experiments.

Many of the defined test bundles and executed experiments have been defined in the context of the case studies presented in Chap. 8.

7.4 Comparative Evaluation of Performance Testing Automation Frameworks

In this section, we compare the proposed approach with the tools we identified as part of the state-of-the-art analysis we present in Sect. 3.7.
Tab. 7.2 we report the list of tools we present in Sect. 3.7 and we additionally add the proposed BenchFlow framework to the list in the *General Purpose Performance Testing Tools (Open-source)* category.
We characterize the tools according to DPE related dimensions we identified as relevant for designing and implementing tools for declarative performance testing integrated with CSDL. The dimensions we evaluate for the presented tools are:

1) *Goal-driven Performance DSL* - whether or not it is possible to specify the performance test objective using a DSL;

2) *Definition of Test Suites* - whether or not it is possible to specify test suites;

3) *Validation of Performance Tests* - whether or not the tool supports for validation of the performance test specification, and if it does at which level of details;

4) *Implement SUT-awareness* - whether or not the tool offers support for specific kinds of software, simplifying the way users can specify performance tests for that software;

5) *Support of Termination Criteria* - whether or not the tool enables the control of the performance test lifecycle with termination criteria;

6) *Automated Scheduling of Performance Tests* - whether or not the tool handles the automated scheduling of performance tests;

7) *SUT Management and Configuration* - whether or not it is possible to manage the SUT and its configuration. This is fundamental for enabling heavy automation;

8) *Data Collection and Metrics* - whether or not the tool manages performance data collection (client-side and/or server-side) and performance analyses computation;

9) *Support of Quality Gates* - whether or not the tool supports the analysis of the test outcome utilizing quality gates defining success or failure conditions for the test.

The table cells mark whether the tool covers the given dimension by using the ✓ symbol, or by reporting details about how the tool covers the given dimension, or if the tools do not cover the dimension we mark the cells by using the ✗ symbol. If the characteristic does not apply to the tool, we use the - symbol.
In the following, we briefly discuss how the different performance testing automation tools cover the different dimensions we identify as relevant and we compare the way each given dimension is supported by the tools, with the way we support the same dimension in the BenchFlow framework.
Goal-driven Performance DSL - To the best of our knowledge, a single tool supports a goal-driven performance test specification, Dynamic Spotter [Sopeco, 2014]. Other tools, dedicated to specific kinds of SUTs implicitly enable goal-driven performance testing targeting specific characteristics of the SUTs. Dynamic Spotter enables specification of exploration performance test goals, where users can state their exploration goal in terms of configurations of the SUT to be explored and metrics of interest to be computed and compared with specified thresholds. Compared to the BenchFlow framework, Dynamic Spotter limits the focus on a specific kind of performance test goal, while in the BenchFlow framework we support multiple performance test goals.
Definition of Test Suites - Some tools, especially commercial ones, enable a test suite specification so the user can collect different performance tests to be executed as a whole. Tools supporting test suite specification enable the selection of tests by referring to files on the file systems specifying the test of interest. In the BenchFlow framework we enable for a richer specification of a test suite, where we enable the user to rely on a declarative specification for selecting subsets of performance test specifications based on labels matching.

Additionally, we provide a mechanism for triggering different performance test suites based on events generated as part of the CSDL.

Validation of Performance Tests - All the identified tools support for performance test specification validation. The precision and accuracy of the validation vary across the different tools. Syntactic validation of the test specification is guaranteed in all of the tools, while semantics validation is present mostly only in tools supporting performance test specifications enabling for semantics validation (e.g., tools relying on custom DSLs). More advanced validation, e.g., validation related to the availability of requested resources as part of the performance test infrastructure, is usually present only in commercial tools. In the BenchFlow framework we support both syntactic and semantic validation, by relying on the declarative DSL and the performance test model we present in Sect. 5.4. We also support more advanced validation, to ensure the execution of performance tests does not fail for unexpected conditions in the performance test or the SUT deployment infrastructure, as discussed in Sect. 5.6.4.

Implement SUT-awareness - General purpose performance testing tools rarely support for SUT-awareness. This feature is mainly present in tools dedicated to performance testing in specific domains, as it is for tools dedicate to middleware and Cloud performance testing. Tools implementing SUT-awareness, often offer test specification languages including entities related to the target SUT domain model, as well as customized metrics relevant to the target SUTs. In the BenchFlow framework and the proposed DSL, we implement SUT-awareness by mainly supporting for simplified workload specification, SUT deployment orchestration, and custom performance metrics.

Support of Termination Criteria - Almost all the commercial tools, and some open-source tools support for termination criteria determining conditions when to interrupt the test execution. This is an important feature for integrating the tool in CSDL and enable continuous performance test execution. Most of the tools supporting termination criteria, enable time-based termination criteria. Some tools also enable termination criteria based on the performance test results quality and the overall quality of the automation process. In the BenchFlow framework we support both time- and quality-based termination criteria and we additionally implement failure-handling mechanisms as part of the framework to identify conditions impacting the test execution and stop it as soon as possible in case of issues impacting the quality of results.

Automated Scheduling of Performance Tests - All the identified tools support for scheduling of multiple performance tests. Most of the tools support queue-based scheduling where submitted performance tests are executed one

after the other. Commercial tools also support for scheduling of performance tests and the performance test infrastructure at the same time. This enables for better scheduling of performance tests and allocation of performance test infrastructure resources. In BenchFlow we support queue-based scheduling of performance tests, where some parallelism is included when enough resources are available. We also take into account the resources available in the test infrastructure for proper scheduling of performance tests.

SUT Management and Configuration - Few general-purpose tools, both open-source and commercial, support for SUT management and configuration. On the other end, most of the tools dedicated to specific types of SUT support this feature, acknowledging the importance of handling the SUT lifecycle for proper performance testing automation. Different tools support SUT management and configuration in different ways, but they mostly rely on standard technologies for deployment descriptor definition, including SUT configuration options. We consider SUT management and configuration support as an important feature for CSDL integration, thus the BenchFlow framework offers a full support for SUT deployment lifecycle and SUT configuration management as part of the test specification.

Data Collection and Metrics - All the tools support performance data collection, fundamental for proper computing of performance metrics. Some tools only support client-side data collection, while others also support server-side data collection. In the BenchFlow framework we support both client- and server-side data collection by providing data collection services that can be configured according to the users' needs.

Support of Quality Gates - Mostly all the tools support the specification of quality gates or quality checks to determine the final result of a performance test. Being this a very important feature for integrating the tool in CSDL, all the tools supporting for CSDL integration also supports quality gates specification. Among them NeoLoad [NeoLoad, 2016] also enables for declarative SLA specification. Quality gates are supported by different tools in different forms, but mostly by enabling SLA requirements to be compared with computed performance metrics. Some tools also support validation of the test results in terms of their overall quality to guarantee the computed performance metrics meet specified quality criteria and can be considered reliable. In the BenchFlow framework we support quality gates based on conditions defining thresholds on all the computed performance metrics, and we ensure computed metrics are reliable by controlling the entire end-to-end lifecycle of the executed performance tests. We also support quality gates for specific test types, for example, regression quality gates.

	Goal-driven Performance DSL	Definition of Test Suites	Validation of Performance Tests	Implement SUT-awareness	Support of Termination Criteria	Automated Scheduling of Performance Tests	SUT Management and Configuration	Data Collection and Metrics	Support of Quality Gates
General Purpose Performance Testing Tools (Open-source)									
BenchFlow[Ferme and Pautasso, 2018]	✓	✓	Syntactic and semantic validation	✓	✓	✓	✓	✓	✓
JMeter[JMeter, 1998]	✗	✗	Syntactic validation	✗	✗	✓	✗	✓	✓
Faban[Faban, 2000]	✗	✗	Syntactic validation	✗	✗	✓	✗	✓	✓
Gatling[Gatling, 2011]	✗	✗	Syntactic validation	✗	✗	✓	✗	✓	✓
PerfCake[PerfCake, 2016]	✗	✗	Syntactic validation	✗	✗	✓	✗	✓	✓
Ascar Pilot[Li et al., 2016]	✗	✗	Syntactic validation	✗	✗	✓	✗	✓	✓
BenchExec[BenchExec, 2015]	✗	✗	Syntactic validation	✓	✗	✓	✓	✓	✓
DataMill[Petkovich et al., 2015]	✗	✗	Syntactic validation	✗	✗	✓	✓	✓	✓
Taurus[Blazemeter, 2014]	✗	✓	Syntactic and semantic validation	✗	✓	✓	✗	✓	✓
Artillery[Shoreditch Ops, 2017]	✗	✗	Syntactic validation	✗	✓	✓	✗	✓	✓
Locust[Locust, 2020]	✗	✗	Syntactic validation	✗	✗	✓	✗	✓	✓
The Grinder[Philip Aston, 2014]	✗	✗	Syntactic validation	✗	✗	✓	✗	✓	✓
Tsung[Nicolas Niclausse, 2017]	✗	✗	Syntactic validation	✗	✗	✓	✗	✓	✓
k6[Load Impact AB, 2019]	✗	✓	Syntactic validation	✗	✓	✓	✓	✓	✓
ElasTest[ElasTest, 2016]	✗	✓	Syntactic validation	✗	✓	✓	✓	✓	✓
Predator[PayU, 2020]	✗	✗	Syntactic validation	✗	✗	✓	✗	✓	✓
AutoPerf[Apte et al., 2017]	✗	✗	Syntactic validation	✗	✗	✓	✓	✓	✓
WS-TaaS[Yan et al., 2012]	✗	✗	Syntactic validation	✗	✗	✓	✗	✓	✓
Weevil[Wang, 2006]	✗	✓	Syntactic and semantic validation	✓	✗	✓	✓	✓	✓
Dynamic Spotter[Sopeco, 2014]	✓	✗	Syntactic and semantic validation	✓	✓	✓	✓	✓	✓
PET[Brad Kemp, 2001]	✗	✗	Syntactic validation	✗	✗	✓	✓	✓	✓
General Purpose Performance Testing Tools (Commercial)									
HP LoadRunner[HP, 2016]	✗	✓	Syntactic validation	✗	✓	✓	✗	✓	✓
NeoLoad[NeoLoad, 2016]	✗	✓	Syntactic and semantic validation	✗	✓	✓	✓	✓	✓
Akamai[Akamai, 2020]	✗	✓	Syntactic validation	✗	✓	✓	✗	✓	✓
Apica[Apica, 2016]	✗	✓	Syntactic validation	✗	✓	✓	✗	✓	✓
Dynatrace's Load[Dynatrace, 2016]	✗	✗	Syntactic and semantic validation	✓	✓	✓	✓	✓	✓
BlazeMeter[Blazemeter, 2016]	✗	✓	Syntactic and semantic validation	✓	✓	✓	✓	✓	✓
WebLOAD[Radview, 2020]	✗	✓	Syntactic validation	✗	✓	✓	✗	✓	✓
LoadComplete[SmartBear, 2016]	✗	✓	Syntactic validation	✗	✓	✓	✗	✓	✓
IBM's Rational Performance[IBM, 2020]	✗	✓	Syntactic validation	✗	✓	✓	✗	✓	✓
LoadStorm[CustomerCentrix LLC, 2020]	✗	✓	Syntactic validation	✗	✓	✓	✗	✓	✓
SilkPerformer[MicroFocus, 2020]	✗	✓	Syntactic validation	✗	✓	✓	✗	✓	✓
RETiT[RETIT GmbH, 2020]	✗	✗	Syntactic validation	✓	✓	✓	✓	✓	✓
BlackFire[Blackfire, 2014]	✗	✗	Syntactic and semantic validation	✓	✓	✓	✗	✓	✓
Middleware Oriented Performance Testing Tools									
SOABench[Bianculli et al., 2010a]	✗	✗	Syntactic and semantic validation	✓	✓	✓	✓	✓	✗
SOArMetrics[Li et al., 2009]	✗	✗	Syntactic and semantic validation	✓	✓	✓	✓	✓	✗
Cloud and/or Cloud-native Oriented Performance Testing Tools (Open-source and Commercial)									
Cloudstone[Sobel et al., 2008]	✗	✗	Syntactic validation	✓	✓	✓	✓	✓	✗
Cloud Work Bench[Scheuner et al., 2014]	✗	✓	Syntactic and semantic validation	✓	✓	✓	✓	✓	✓
Smart CloudBench[Chhetri et al., 2016]	✗	✗	Syntactic validation	✓	✓	✓	✓	✓	✓
cbtool[Silva and Hines, 2016]	✗	✓	Syntactic and semantic validation	✓	✓	✓	✓	✓	✓
CloudPerf[Michael et al., 2017]	✗	✗	Syntactic validation	✓	✗	✓	✓	✓	✓
CloudTest[Podelko, 2016]	✗	✗	Syntactic validation	✓	✗	✓	✓	✓	✓
radon-ctt[RADON, 2020]	✗	✓	Syntactic and semantic validation	✓	✓	✓	✓	✓	✓
PerfKitBenchmarker[Google Cloud Platform, 2020]	✗	✗	Syntactic validation	✓	✗	✓	✗	✓	✗

Table 7.2. Performance Testing Tools and DPE Dimensions in CSDL

7.5 Concluding Remarks

In this chapter, we present the different evaluations we performed to assess and validate our claims, according to the R.G.s we define in Sect. 1.2.1. We perform an expert review to assess the overall proposed approach with experts from the performance engineering domain, both from academia and industry. In the expert review, we particularly focus on assessing the expressiveness, effort, and suitability of the proposed approach. We perform a summative evaluation with target users of our approach, to assess the learnability of the proposed DSL. In both the proposed surveys we also ask participants to compare the proposed approach with widely-used standard imperative approaches. The results of the expert review and the summative evaluation indicate the proposed declarative DSL we defined as an answer to R.G. 2 and R.G. 4 is suitable to support the target users in specifying performance tests to be integrated with CSDL, and it is considered more suitable than standard imperative approaches in supporting them to do so. The participants also provide constructive feedback to improve the approach and indicate they are willing to use the framework implementing the proposed approach for their performance testing activities.
We describe the iterations the proposed BenchFlow framework underwent from its initial prototype to the current version in an iterative review, where we also report data on the number of people collaborating on the framework development, and the number of experiments and performance data analysis we performed with the framework over the years. We also compare the BenchFlow framework with other performance testing solutions available on the market concerning dimensions we defined as relevant for frameworks implementing a declarative performance testing automation approach in CSDL. The BenchFlow framework, that has still to be considered a research prototype and far from being a complete performance testing solution, is the first framework supporting a completely declarative approach for performance testing, according to our analysis. The multiple collaboration with other researchers, also validate the solution we propose as an answer to R.G. 3 as suitable for performance testing automation.

Chapter 8

Case Studies

In this chapter, we present the main case studies in which we developed, improved, and applied the proposed approach. We present four case studies in total. The first one present the work is done in the context of benchmarking BPMN 2.0 WfMSs, the first system we targeted with our approach. We then present a second case study in which we integrated the BenchFlow framework we propose with another framework in the context of performance testing automation in CSDL. The last two presented case studies are about adding a new layer of abstraction on top of the DSL proposed in this dissertation and about defining a performance-based domain metric for Microservices deployment architectures assessment. For each case study, we highlight the work done, and how it is related to the approach we propose in this dissertation. We refer to published work for additional details on the case studies.

8.1 Benchmarking Workflow Management Systems

Workflow Management Systems (WfMSs) enable users to schedule, execute, and manage the execution of business processes [Specification, 1999; Leymann and Roller, 2000]. As defined by the Workflow Management Coalition (WfMC), the WfMS is "a system that completely defines, manages, and executes 'workflows' through the execution of software whose order of execution is driven by a computer representation of the workflow logic" [Hollingsworth, 1995]. Most of the WfMSs available on the market also allow users to gain insights on executed processes, interact with the control and data flow and build views to facilitate the work of different users to interact with such systems. WfMSs

are designed by exposing APIs offering process management functionalities to other systems and are often integrated with other systems to support business process execution.

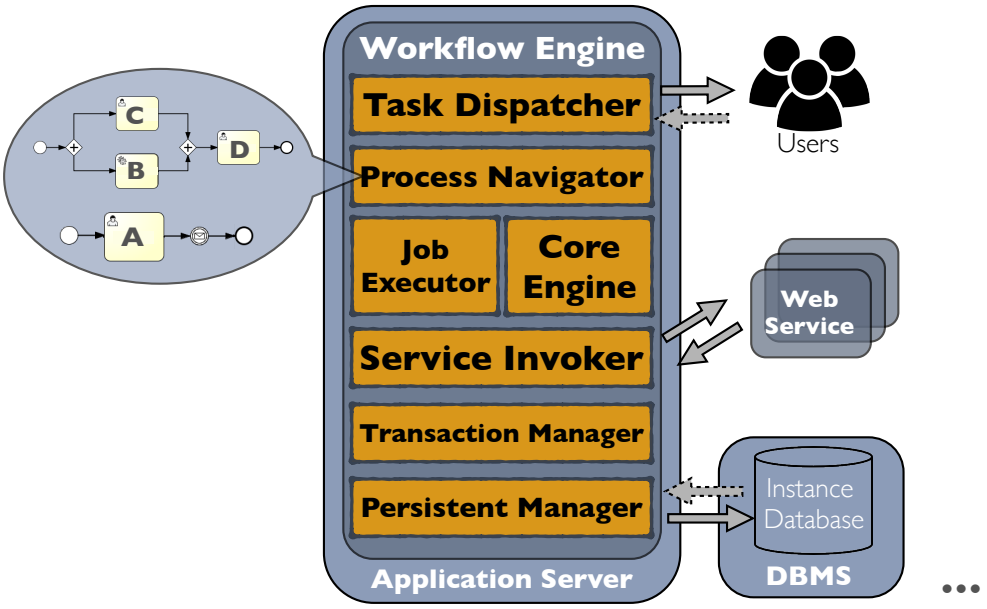

Figure 8.1. WfMSs Main Internal Components

In Fig. 8.1 we present an overview of the common internal elements and services of a WfMSs. The core internal service of a WfMSs is the workflow engine. The workflow engine is responsible for accepting requests from the interacting users, being real users or other systems, and dispatching (*Task Dispatcher*) the received requests to internal services for execution coordinated by the *Core Engine* service. To execute the processes, the workflow engine relies on the *Process Navigator* to navigate the control and data flow of business processes, and according to the definition different internal services are involved. The *Job Executor* is involved when asynchronous tasks have to be executed (e.g., tasks based on time constraints). The *Service Invoker* is involved every time the executed business process invokes an external service, usually via REST APIs. The *Transaction Manager* and the *Persistent Manager* services are always involved for all the business processes executions, being responsible for persistently store the business process state on dedicated DBMS.

We started our work in performance testing automation, contributing to the BenchFlow research project aiming at designing the first benchmark for assessing and comparing the performance of WfMSs. As evident from Fig. 8.1, WfMSs are complex systems, and a benchmark to assess their performance is needed for proper comparison of different solutions available on the market. The BenchFlow research project focused in particular on BPMN 2.0 WfMSs, due to rise of number of engines supporting BPMN 2.0 as the standard modeling language for business processes [Skouradaki et al., 2015b].
In the context of BPMN 2.0 WfMSs we contribute:

1) preliminary analysis of all the challenges involved in benchmarking WfMSs, both from the benchmark procedural side [Pautasso et al., 2015] and from the technical point of view [Skouradaki et al., 2014]. Starting from the challenges we identified, we proposed a research plan towards providing the first benchmark for BPMN 2.0 WfMSs [Skouradaki et al., 2015b].

2) a framework for benchmarking of WfMSs [Ferme et al., 2015]. As part of the technical challenges, we also identified the need of automating the performance experiments needed for exploring the performance of WfMSs and defining a benchmark. This contribution represents the first version of the BenchFlow framework presented in Chap. 6;

3) a methodology for benchmarking WfMSs [Ferme et al., 2016b] accounting for the complexity of such activities, and for the need of properly involving the companies releasing the WfMSs included in the benchmark. Benchmarks need to be repeatable, representative, portable, scalable, relevant, efficient, accessible, affordable, and simple [Kounev et al., 2020; Huppler, 2009; Gray, 1992; Sim and Easterbrook, 2003; Brebner et al., 2005; Crolotte, 2009]. The proposed methodology offers solutions to both technical and logistic challenges in benchmarking WfMSs. The research performance in defining the proposed benchmarking methodology provided us with feedback influencing the design of the BenchFlow framework, e.g., concerning the need for transparency and reproducibility;

4) the first benchmark for assessing and comparing the performance of BPMN 2.0 WfMSs, relying on workflow patterns [Skouradaki et al., 2016]. As part of this work we define the initial versions of the DSL we propose in this dissertation and we heavily rely on the BenchFlow framework for automating the experiment execution and disseminating the benchmark

results. We refer to the automation-oriented performance tests catalog presented in Chap. 4 for identifying the different performance tests to be executed to properly assess the performance of the SUT;

5) an analysis of the performance of BPMN 2.0 WfMSs when considering Public Cloud execution environments [Ferme et al., 2016a], and performance comparison of different versions of the same BPMN 2.0 WfMSs to assess how the performance of such systems under the same workload evolve [Ferme et al., 2016a]. We rely on the automation capabilities of the proposed approach for declaratively defining and executing performance experiments. We rely on the BenchFlow framework also for assessing the performance overhead of different performance modeling practices [Ivanchikj et al., 2017] and in evaluating multi-tenant live migration effects on performance for a BPMN 2.0 WfMSs configured for multiple tenants [Rosinosky et al., 2018].

We summarize most of the work we did in providing the first benchmark for BPMN 2.0 WfMSs in a book chapter [Ferme et al., 2019], and in two research work discussing the lessons learned in evaluation WfMSs performance [Lenhard et al., 2017; Ferme et al., 2017a], as well as part of a pattern language we contribute describing different patterns to consider when approaching WfMSs conformance and performance benchmarking [Harrer et al., 2017]. In the book chapter [Ferme et al., 2019] we also summarize all the performance metrics and KPIs we define for benchmarking BPMN 2.0 WfMSs, presented in Sect. 6.3.5 and provide complete examples.

In the context of the work performed for assessing and comparing the performance of BPMN 2.0 WfMSs we started to contribute towards the DPE approach for performance testing automation presented in this thesis. Particularly relevant for the case of benchmarking BPMN 2.0 WfMSs is the support for rich and expressive control of performance tests over different configurations of the SUT, and the possibility to schedule a large number of tests with different goals. The declarative DSL is very relevant for sharing the research artifacts with other researchers, because the defined tests are easily readable also by not expert people in the performance engineering domain. Additionally, the possibility to distribute test bundles, collect performance data, and compute metrics and statistics is also very important.

As evident from the number of publications, benchmarking BPMN 2.0 WfMSs has been one of the main research areas where we applied the approach we propose in this dissertation and one of the main drivers. Based on our experience we consider the proposed DPE approach as an enabler of performance research

work heavily based on experimentation because it allows researchers to focus on the core of their research goals, relying on a complete solution for defining and scheduling the performance tests needed for validating their assumption. Additionally, by relying on the expressiveness of the declarative specification of performance tests, we found that disseminating the research is also facilitated by the easiness to understand the test specifications.

8.2 Integration of the BenchFlow Approach with the ContinuITy Approach

ContinuITy is a project focusing on ensuring automated, efficient, and sustainable load testing by using continuously recorded measurement data from systems operating in production. ContinuITy also focuses on integrating load testing into continuous software development, and providing a mechanism to (semi-)automatically evolve performance test definitions across multiple versions of the SUT.
We had the opportunity to strictly collaborate with researchers working on the ContinuITy project during our internship based on the SNSF project "Declarative Continuous Performance Testing for Microservices in DevOps"[1]. During the collaboration, we integrated ContinuITy, the tool developed as part of the ContinuITy project, and the BenchFlow framework for extending the capabilities of both tools. ContinuITy provides a Web-based framework for continuously collecting performance data from systems deployed in production, and APIs to generate load tests starting from such collected performance data. The BenchFlow framework on the other end offers a powerful solution for performance testing automation based on a declarative specification of performance tests.
In Fig. 8.2 we present the overview of the approach resulting from the integration of the ContinuITy tool capabilities with the ones of the BenchFlow framework. We merged the capabilities of both the approaches and enabled the possibility for the users to specify declarative performance goals using the BenchFlow DSL, optionally omitting details related to the *workloads*, and the *Load Function*. Omitted details are populated by the ContinuITy tool relying on operational profiles of the system running in production and provided SUT APIs specification codified using Swagger[2]. The user can configure how

[1] http://p3.snf.ch/project-178653, last visited on February 7, 2021
[2] https://www.swagger.io/, last visited on February 7, 2021

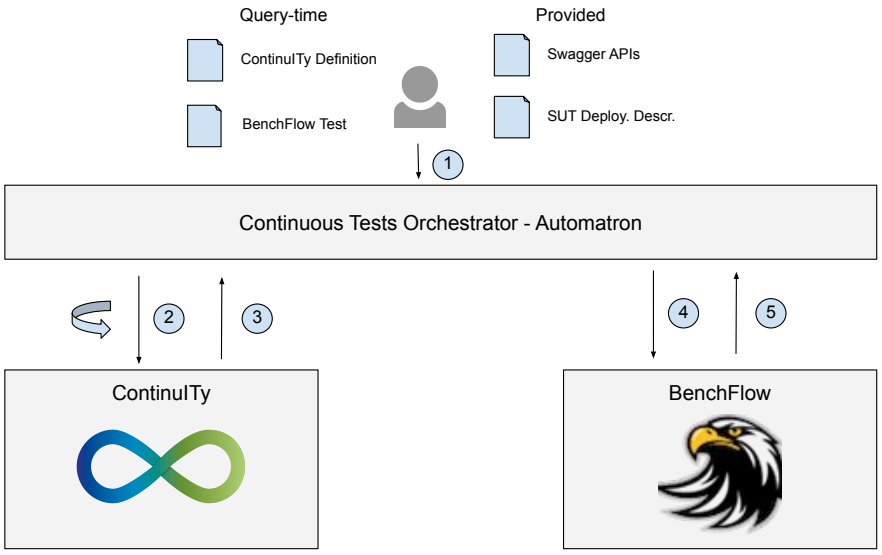

Figure 8.2. ContinuITy and BenchFlow Framework Integration

to retrieve such data relying on a ContinuITy definition file or relying on defaults provided by the integration layer we named *Automatron*. The provided integrated solution enables users to request, for example, to execute a regression test defined using the BenchFlow DSL without specifying the *workloads*. ContinuITy is then interrogated by the *Automatron* integration layer and requested to populate the BenchFlow test specification with *workloads* related data. After populating the *workloads* the test is ready to be scheduled for execution on the BenchFlow framework. The *Automatron* integration layer is designed as a CLI so that it can be integrated into tools part of the CSDL. The integration between the ContinuITy tool and the BenchFlow framework was developed as part of a Master thesis [Palenga, 2018], and it is available as part of the open-source code of the *continuity* tool on GitHub[3].
This case study presents a first extension on the proposed declarative approach for performance testing automation in CSDL. Empowering the expressiveness

[3] https://github.com/ContinuITy-Project/ContinuITy/tree/master/continuity.service.benchflow, last visited on February 7, 2021

of the BenchFlow DSL, and the APIs and the scheduling and automation capabilities of the BenchFlow framework we managed to improve the overall DPE approach by simplifying, even more, the work for users defining performance test.
This contribution is important, especially because one of the feedback we received as part of the expert review mentioned the complexity of the workload specification.

8.3 Behavioral-driven Performance Testing

Behavioral-driven testing enables users to specify the expected behavior of a system using controlled natural languages. The focus of behavioral-driven testing is on the business requirements of the implemented solution, rather than the implementation of the same.
Our target users, although indicating the DSL and the overall approach as good in terms of learnability and usability, also indicated additional abstraction on the performance test specification might be useful (Sect. 7.2.6).
We collaborated with other researchers to offer an additional abstraction on top of the declarative DSL we propose as part of this dissertation. We designed and implemented a Behavior-driven Load Testing (BDLT) language [Schulz et al., 2019] allowing users to specify: a) a performance test request in the form a controlled natural language relying on the *GIVEN*, *WHEN*, *THEN* keywords specifying the expected condition of the SUT or its deployment context before starting the performance test, the configuration of the SUT and the actual load to issue to the SUT respectively; b) a time reference to the operational profile of a deployed system, to be used as a reference point to generate load tests using operational data. Compared to the different types of tests supported by the BenchFlow DSL, in this case study we mainly focused on load tests and exploration tests.
The BDLT specification is converted for execution to a BenchFlow performance test specification by relying on the integration layer built to integrate the ContinuITy tool and the BenchFlow framework presented in Sect. 8.2. In Fig. 8.3 we represent the mapping between the elements of a BDLT specification and the BenchFlow DSL. The conversion happens by relying on template specifications provided by the BenchFlow framework and populated with data coming from the BDLT specification provided by the user. More details on the BDLT specification, the mapping to the BenchFlow DSL and real-world case studies are provided in [Schulz et al., 2019].

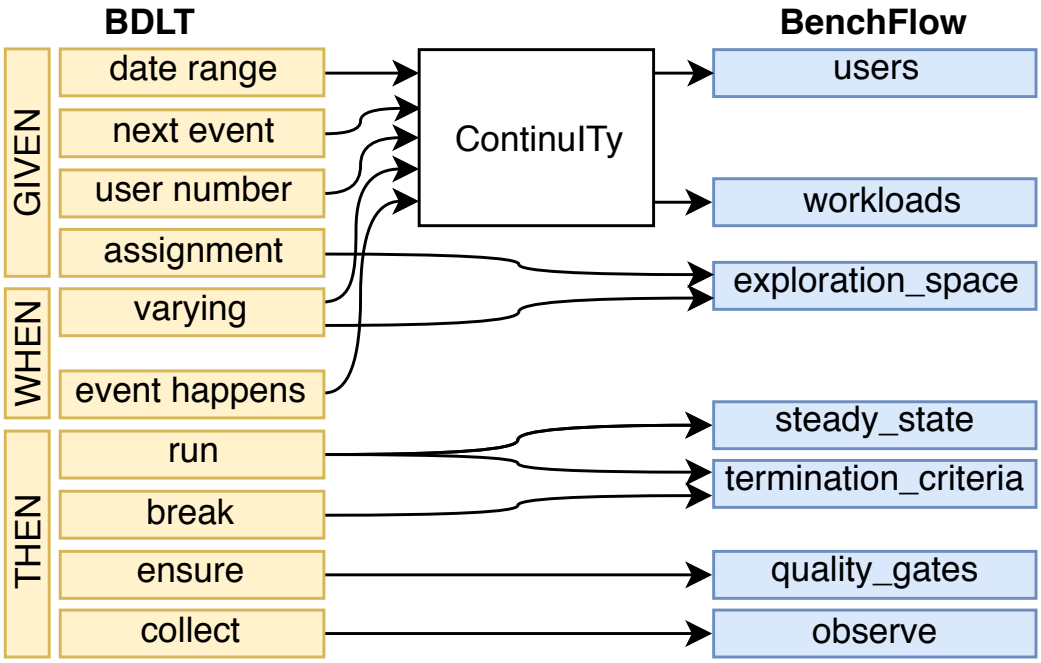

Figure 8.3. Behavior-driven Load Testing Mapping to BenchFlow DSL [Schulz et al., 2019]

As part of the same research work, we also collaborated on defining an approach for generating performance reports based on the intent the user specifies as part of the test [Okanović et al., 2019]. In Fig. 8.4 we present an overview of the overall approach. The report generation takes into account the performance intent specified by the user, e.g., by relying on the BDLT specification or the BenchFlow DSL, filters the collected performance data and then select proper visualization techniques to best answer the performance query specified by the user. More details on the overall approach are discussed in [Okanović et al., 2019; Zorn, 2018].

This case study works on abstracting, even more, the language exposed to end-users of the declarative performance testing automation approach we propose. With this case study, we also face one on the feedback received as part of the summative evaluation, indicating the opportunity to abstract even more the specification language.

The proposed abstraction enables users to use a controlled natural language to express their performance intents, but limits the control and expressiveness compared to the DSL proposed in this dissertation.

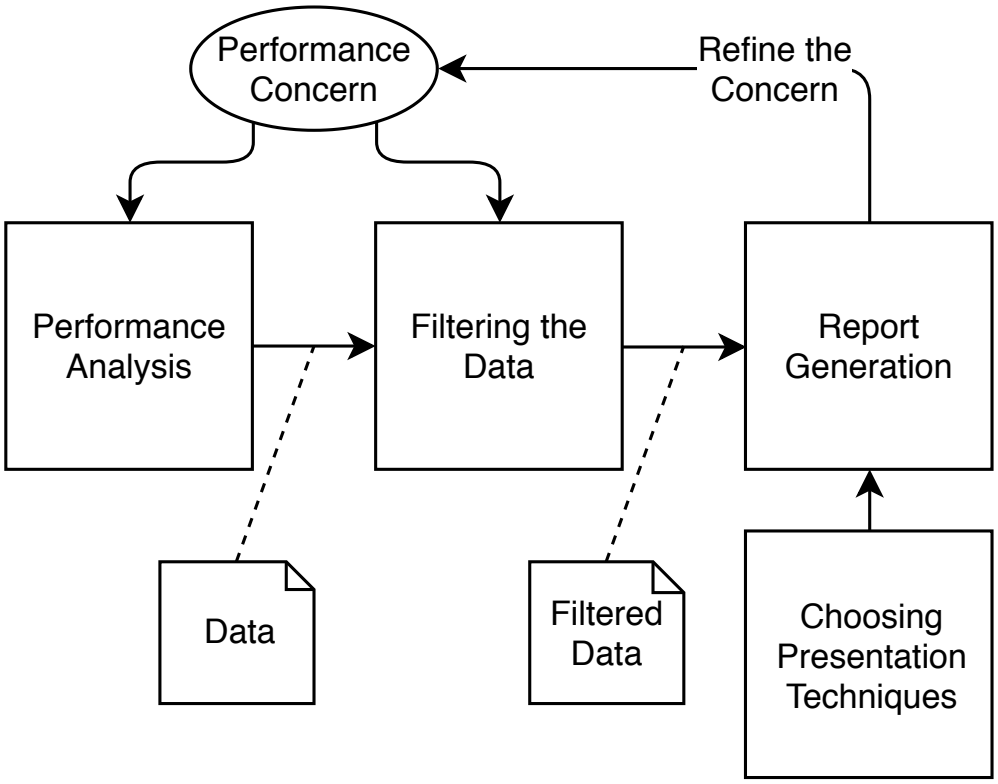

Figure 8.4. Concern-driven Reporting of Software Performance Analysis Results [Okanović et al., 2019]

8.4 Definition of a Performance-based Domain Metric for Services' Deployment

Microservices can be deployed in different deployment contexts and by utilizing different deployment architectures.
Assessing the different deployment architectures in terms of performance according to the expected workload the system is going to experience, is a complex task. On the other end, it is important to know the most suitable deployment architecture for a workload so that the system can evolve and adapt properly to the load by optimally utilizing the underlying computation resources.
We collaborated with other researchers to design a novel approach for Microservices deployment architecture assessments by defining a performance-based domain metric [Avritzer et al., 2020, 2018]. We fully automatized the proposed approach in a tool [Avritzer et al., 2019] relying on the ContinuITy

tool and the BenchFlow framework for performance tests definition and execution. The proposed approach automatically executes a baseline performance test to measure the performance of the SUT and then tests any given deployment configuration with different user profiles derived from production usage of the SUT. The result of the test is a normalized domain metric indicating how well a given configuration supports the issued load, defined by applying a scalability requirement on the results obtained from each test when compared to the baseline. In Fig. 8.5 we present an overview of the proposed approach. The approach consists of the following steps:

1) *Collection of operational data*: data about the system performance are collected from production environments where the system is deployed. The collected data usually refers to the number of received requests over a given period. For this step, we rely on the ContinuITy tool capabilities to collect and model operational data. The data are stored on InfluxData[4], a time-series data store;

2) *Analysis of operational data*: after collecting the operational data from production, the data are analyzed relying on scripts we built on top of Jupyter notebooks[5]. In this step, we estimate the probability of the occurrence of a certain workload situation. For example, when collecting the number of requests per second over a given period, in this step we estimate the probability of a given number of users interacting with the SUT at any given point in time. To aggregate the data we compute the number of interacting users on a given moving time window;

3) *Experiment generation*: we rely on the computer workload situations and data modeled by the ContinuITy tool to generate BenchFlow tests starting from provided templates. The generated BenchFlow tests target different deployment architectures of the SUT and are executed for different workload situations. A baseline performance test is always executed before the other performance tests, to assess the baseline performance of the SUT;

4) *Experiment execution*: generated experiments are executed relying on the BenchFlow framework and performance data are collected. The BenchFlow framework computes metrics of interest as well, and then we compute the results of the experiment execution by comparing the results

[4] https://www.influxdata.com/, last visited on February 7, 2021
[5] https://jupyter.org/, last visited on February 7, 2021

obtained for each configuration with the baseline, by assessing a defined scalability requirement for the SUT. An example of scalability requirements is that across the different workload situation the response time of APIs targeted during the test have to be within a given range from the one measured in the baseline test;

5) *Domain metric calculation*: starting from the experiment results, we compute a domain metric normalized between 0 and 1. The domain metric represents how well a given configuration supports the workload situations, by taking into account how many violations of the defined scalability requirement the given configuration experiences.

We applied the developed approach to several deployment configurations of a Microservice application, handling the performance test specification and performance test execution using the BenchFlow framework. The experiments we performed as part of our work [Avritzer et al., 2020] showed that careful performance assessment of Microservices deployment is important to properly determine to which service allocate resources, and how many resources to allocate.
This case study is related to the thesis because it has been enabled by the automation capabilities offered by the BenchFlow framework and the expressiveness and reusability of the DSL.
The researchers we collaborated with for this case study appreciated the expressiveness of the declarative DSL and the opportunity to schedule a large number of complex performance tests to define a novel approach for Microservices deployment architecture assessment.

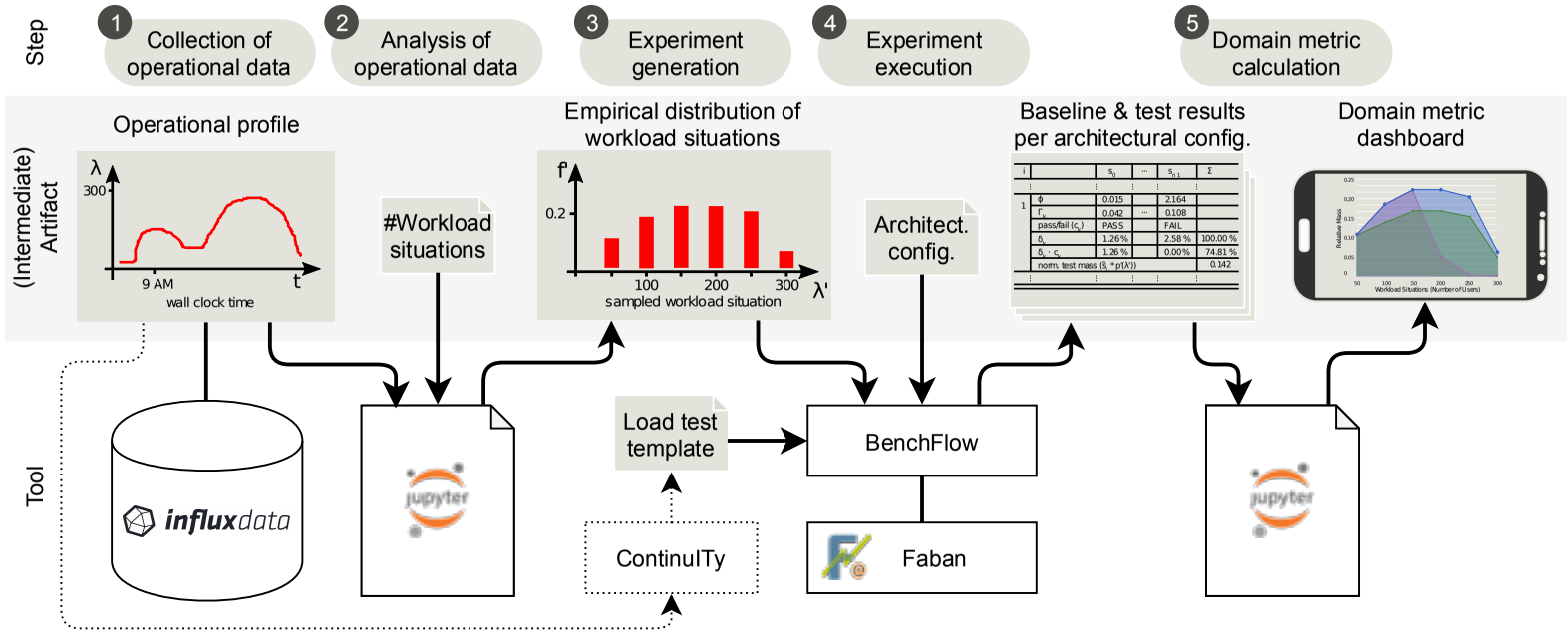

Figure 8.5. Scalability Assessment of Microservice Architecture Deployment Configurations [Avritzer et al., 2020]

8.5 Concluding Remarks

We had the opportunity to collaborate with many researchers around the World. During such collaborations, we had the opportunity to develop and improve the approach we propose in this dissertation, and to apply it in different contexts. In this chapter, we present the most relevant case studies we performed collaborating with other researchers and how they are related to the DPE approach for performance testing automation we propose, other than how they contributed to extend and improve the proposed approach. Across the different case studies, we relied on the proposed approach for automating performance tests execution for different kind of systems, and for defining performance test templates partially populated by other frameworks, to simplify the work of users defining declarative performance tests. In one case study, we also contributed a controlled natural language for a behavioral-driven performance test definition. The controlled natural language represents an abstraction on top of the DSL part of this thesis and is mapped to our DSL for executing the specified performance tests.

Part IV

Epilogue

Chapter 9

Lessons Learned

In this chapter, we present an overview of lessons learned while designing and developing the first DPE approach for continuous and automated execution of performance tests alongside the CSDL. The proposed approach builds on the great work done in collaboration with other researchers, and students to design and develop the DSL and the BenchFlow framework. Across different collaborations we had the opportunity to support other people in deploying the proposed approach in their research groups, we extended the proposed approach to support multiple systems and for integrating it in third-party systems, and we iteratively developed and improved the proposed approach continuously including feedback received from other researchers.

9.1 Lessons Learned in other People Using the BenchFlow DSL and Framework

We deployed the BenchFlow framework in different contexts and research groups. In some research groups, other researchers took care of deploying the framework, in other instances we took care of the framework deployment part. Relying on Docker for developing and deploying the proposed framework helped us in simplifying the way the framework itself can be deployed and operated, this simplified the work of other researchers deploying the framework in their research labs as well. Some basic knowledge of Docker is required, but most of the activities are codified in declarative deployment descriptors that can just be executed from other people via the BenchFlow CLI, assuming at least an average understanding of infrastructure operations and Unix operating system. In many cases, most of the actual users of the BenchFlow

framework in other research groups were defining performance tests relying on the DSL. They used the framework as the target users of the summative evaluation we propose in Sect. 7.2. What we learned from people using the framework, with whom we had direct collaboration and discussions, allowed us to improve the overall proposed approach over different iterations. The first version of the proposed approach and the BenchFlow framework had limited usability, but the latest version has been evaluated much better when compared to previous versions in terms of expressiveness and overall usability of the proposed approach. The most complex part we had to deal with has been the non-deterministic behavior of the SUT deployment, execution, and undeployment. We collected much feedback from on-the-field experiments and we codified the knowledge as part of the BenchFlow framework to provide the users with facilities and automation to manage the SUT lifecycle. Although the learning curve to adopt the DSL has been considered average, the result obtained by the automation opportunities and the possibility to reuse specifications across multiple experiments has been considered a key advantage by all the researchers adopting the proposed approach. People we collaborated with especially appreciate the possibility to define performance test specifications that can also easily be presented to other people and/or in research papers and can then be used to automate performance tests execution and reused across many different experimentation settings. Being mostly researchers, another big advantage people we collaborated with appreciated, is the possibility to distribute test bundles, collected performance data, and computed metrics as part of the artifacts produced in their research work in an easy way. By providing the test bundles, other researchers can reproduce the same experiments by relying on the BenchFlow framework.

9.2 Lessons Learned in Extending the BenchFlow DSL and Framework

The different versions of the BenchFlow DSL and framework have been developed with the contributions of different people, mainly computer science students collaborating with us while developing their Master theses. The complexity and the distributed nature of the framework required the proper approach for successful management of the project with multiple contributing people, since the initial phase of development. After an initial prototype of the BenchFlow framework and the DSL was developed, we started to organize the

project as an open-source project on GitHub[1] and we applied DevOps practices and developed the different iterations of the proposed approach following a CSDL. The main advantages of such an approach for us have been the possibility to set and share a standard development process for different students contributing to the project, automate quality enforcement and structure the code review process to validate the quality of the contributed improvements. Applying these practices in our research project helped us maintaining control of the quality of the developed solutions, and offered the students contributing to the development the opportunity to learn and follow a development and delivery approach quite common in the industry.
The different functionalities of the BenchFlow framework required us to rely on multiple software development languages and third-party solutions. We decided to develop the BenchFlow framework on multiple cohesive services collaborating to offer the overall functionality. To ensure the provided solutions kept working over the different development iterations, we automated as much as possible unit, system, and end-to-end functional tests [Clemson, Toby, 2014]. We also automated performance tests for some of the BenchFlow framework APIs by relying on the BenchFlow framework itself in the latest stable version. During the development of the BenchFlow framework, we learned the importance of selecting widely supported programming languages, providing support for proper testing. We as well realized the importance of relying on open-source software with strong development communities and good support, to successfully integrate third-party solutions and define automated tests for the integration with the same. For more details on the development process, we followed refer to [Findahl, 2017, Appendix A]

9.3 Lessons Learned in Integrating the BenchFlow DSL and Framework with other Systems

We integrated the BenchFlow with other frameworks, as discussed in Sect. 8.2, and Sect. 8.3. We as well integrated the BenchFlow framework with CI tools, relying on the provided CLI. In the case of the integration performed in Sect. 8.2, we leveraged the DSL library for generating performance tests relying on defined starting template specifications and customizing such specifications for the different needs we had during experimentation. In both cases the integration between the third-party tool and the BenchFlow framework relied on

[1] https://github.com/benchflow/, last visited on February 7, 2021

the RESTful APIs exposed by the BenchFlow framework. In the case of the integration with the ContinuITy framework (Sect. 8.2) we also relied on the DSL model exposed using the DSL library for automatically generate performance tests and workloads by relying on the capabilities of the ContinuITy framework. The BenchFlow framework RESTful APIs and the DSL library have been key for supporting a successful integration with other frameworks and solutions. They enabled the opportunity to rely on the proposed approach capabilities to provide additional functionalities and solutions in the context of DPE when applied to performance testing. Relevant APIs in this context have been also the ones dedicated to monitoring the execution state of the scheduled performance tests and the rich lifecycle management offered by the BenchFlow framework that can be leveraged by integrating frameworks to fine-tune their execution lifecycle.
For what concerns the work presented in Sect. 8.3, we mainly leveraged the DSL library as the tool used for translating a Behavior-driven Development (BDD) testing definition to actual executable performance tests scheduled by the BenchFlow framework. For integrating the proposed solution with other approaches for modeling declarative performance tests, the DSL library proved to be a great enabler.
When integrating the BenchFlow framework with CI tools, we mainly relayed on the CLI for the integration. Although the APIs provide a great interface for accessing BenchFlow framework's functionalities, the CLI allowed us to codify the interactions with the APIs, results, and error handling in a CI tool-agnostic way. This has the advantage of offering the opportunity to integrate the proposed approach in most of the available tools in the DevOps context, by embedding the BenchFlow CLI as part of the tools' solutions. We acknowledge the advantages, especially in terms of user experience, of developing dedicated integration with different tools, e.g., relying on plugins, but we learned that this can happen later in the integration process. The CLI provides a fast and reliable way to enable the integration with limited effort.

Chapter 10

Conclusion

In this chapter, we summarize our contributions and discuss threats to validity for the evaluations we performed.

10.1 Thesis Summary

In this work we present the first DPE approach for continuous and automated execution of performance tests alongside the CSDL. The proposed approach allows real users and systems to automate performance tests specified relying on a proposed declarative language, and integrate them in CICD pipelines.
Compared to the state-of-the-art in traditional performance testing, declarative performance testing, and automation of performance tests in CSDL, we contribute a novel end-to-end approach for declarative performance test specification, allowing real users and systems to state the test goal and control the entire automation process to achieve the stated goal.
The target systems of our approach are mainly RESTful Web services and BPMN 2.0 WfMSs, while the target users are practitioners involved in the CSDL, and in particular developers, software testers, quality assurance engineers, DevOps engineer, and operations engineers.
To enable real users and third-party systems to declaratively specifying performance test automation processes, we developed an expressive DSL enabling the specification of goal-driven performance tests and their end-to-end automation process. Users of the DSL can:

a) define many different types of performance tests and test suites having different goals,

b) validate the test and test suite specification,

c) define the workloads to be issued to the SUT and how to mix workloads' operations,

d) define the performance metrics and statistics of interest and performance data to be collected,

e) configure the process to be executed to achieve the stated goal,

f) control the execution of the test process and set criteria about its termination,

g) define criteria determining when a test or test suite is successful or failing,

h) define the SUT and its services configuration,

i) as well as define CSDL events activating the different test suites.

The proposed DSL allows a clear and validated specification of the performance test goal, and facilitate reuse of performance test specifications. Along with the DSL we provide many examples and template specifications the users can rely on to specify performance tests having different goals extracted from an automation-oriented performance tests catalog we propose. We also describe a proposal the users of the DSL can refer to when defining the different types of tests to be integrated in different moments of the CSDL according to the events generated as part of the process.
To support users and third-party systems in automating the execution of tests specified using the proposed DSL, we develop a model-driven framework, named BenchFlow, configured using such DSL and driving the entire end-to-end performance test automation process according to the test specification. The framework contributes to different services, enabling the execution of declaratively specified tests. The main services are:

a) validation of declarative performance tests and test suites,

b) scheduling of declarative performance tests and test suites,

c) automatic execution of the scheduled tests,

d) automatic generation of load drivers needed for the test execution,

e) automatic lifecycle management of the SUT including deploying, configuration and undeploying,

f) automated performance data collection,

g) automated metrics and statistics computation for different SUT types.

Relying on the DSL and the BenchFlow framework, users and third-party systems can rely on a complete DPE approach for continuous and automated execution of performance tests alongside the CSDL. Users and third-party systems can declaratively specify performance tests and test suites and their automation process using the DSL, and submit them for execution to the BenchFlow framework. They can access the status of the execution at any point in time, relying on the provided RESTful APIs and the provided CLI, and interact with the execution lifecycle to provide for input or to stop the execution. At the end of the test execution, when test results are available, users and systems can also access metrics and statistics about the performance of the SUT related to the tests. They also get access to all the collected performance data and artifacts generated for, during, and after the automated execution of the performance test.
Empowering the library implementing the DSL, the BenchFlow framework APIs and the provided CLI, users can extend the approach as for example by: a) integrating the library in a tool part of the CSDL generating performance tests according to observed conditions and submitting them to the BenchFlow framework for execution; b) integrating the CLI in processes defined in tools part of the SUT deployment environment, for example in an APM, for scheduling performance tests and performance test suites according to experienced performance behavior to ensure performance requirements codified in the test specifications are still valid.
With the proposed approach for continuous and automated execution of performance tests alongside the CSDL we aim at contributing to facilitating the integration of performance activities alongside the CSDL and at enabling the target users of our approach to specifying performance tests. To validate our proposal we iteratively improved the proposed approach by applying it in different use-cases heavily requiring performance test automation, as well as by performing an expert review and a summative evaluation of the DSL and the proposed approach. The results of the survey corroborate our proposition and confirm the proposed approach is valid for the aims it has been built for. In particular, from the expert review, we learned that the expressiveness of the DSL is considered on the median as *good* in covering the needs of the target users and suitable for them for the activities of specifying and executing performance tests. The reusability, the suitability of the approach for the target users, the perceived usability and effort is considered on median *much better* (the average, if existing, would lie in between of *somewhat better* to *much*

better) than standard approaches for performance testing, mostly imperative ones. Participants of the summative evaluation confirmed the DSL is easy to learn and understand. After a short introduction to the approach and all the elements part of the DSL, more than 80.0% of the participants on average correctly answered the proposed multi-choice tasks evaluating the usability and reusability of the language elements. This held also when the participants had limited to no knowledge of the performance testing domain. Participants also confirmed the expressiveness and reusability of the DSL is *good* for expressing their performance testing needs, and they particularly appreciate the goal-driven specification of tests. As target users, they also confirmed they would prefer the proposed approach to implement performance tests, compared to standard imperative approaches and that they are interested in using the framework implementing the proposed approach.

10.2 Summary of Contributions

To answer the research goals we state in Sect. 1.2.1, we contribute:

1) An automation-oriented performance tests catalog for DPE in Chap. 4;

2) An analysis of how to integrate different types of performance tests identified in the proposed catalog as part of the CSDL in Sect. 4.4;

3) A DSL for declarative specification of performance test automation process in Chap. 5. We evaluate the proposed DSL usability in terms of learnability and reusability, by performing an expert review and a summative evaluation presented in Sect. 7.1 and Sect. 7.2 respectively;

4) A framework for declarative performance testing in Chap. 6. We evaluate the proposed framework, by performing an iterative review and a comparative evaluation presented in Sect. 7.3 and Sect. 7.4 respectively.

10.3 Threats to Validity

The main threats to the validity of our work refer to the evaluation of the same. We also mention threats to validity related to the implemented DSL and the BenchFlow framework.

10.3.1 Construct Validity

Threats to construct validity are related to the relationship between theory and observation. In our work, we have threats to construct validity for the surveys we used to evaluate the proposed approach. We proposed the users a multi-choice tasks based survey, where we requested them to select the correct answers, as well as we proposed to evaluate some statements using a Likert scale. We mitigate threats to construct validity by relying on standardized evaluation methods, and standardized Likert's scale options to collect participants' feedback, and we asked participants to express additional feedback using free-text fields to report their opinion on the different answers they provide.

10.3.2 Internal Validity

Threats to internal validity are related to factors that could have influenced the results. In our work, the surveys we relied on to evaluate the proposed approach are affected by internal validity. We mitigate internal validity threats, by collecting knowledge from a diverse set of participants reached with different media, with different prior knowledge on the concepts relevant for the evaluation and we assess such by collecting data on the knowledge of all the participants before proposing them the survey.

10.3.3 Conclusion Validity

Threats to conclusion validity concern factors leading to reach an incorrect conclusion about a relationship in the observations. In our work, the comparative evaluation we performed including the overall proposed DPE approach and other performance testing tools experience threats to conclusion validity. We conclude there are no other approaches available allowing declarative and automated execution of performance tests alongside the CSDL. We mitigate the risk of missing available solutions by performing an in-depth analysis of state-of-the-art and commercially available solutions, and by asking in both proposed surveys about similar approaches to the proposed one, known by participants.

10.3.4 External Validity

Threats to external validity concern the generalizability of our findings. We mitigate the threats to external validity in the findings we obtained with the proposed surveys, by targeting specific groups of diversified professionals. The

expert review has been performed targeting experts in the performance testing domain, while the summative evaluation involved both professionals and students, with different degrees of experience. We claim the study provides good coverage of the potential categories of target users, although further studies with more participants and on the field are advisable due to the limited number of participants (63 for the summative evaluation, and 18 for the expert review). In terms of the proposed tasks in the summative evaluation, we selected tasks covering all the elements of the test specification we propose in the DSL, and we randomized answers' options for each task for each participant to avoid possible patterns in identifying the correct answers. However, we can not exclude our results depend on the particular choice of the tasks. We implemented the DSL and the BenchFlow framework by mainly looking at two types of systems: RESTful Web services and BPMN 2.0 WfMSs. This introduces threats to external validity because the design and implementation decisions might not apply to other kinds of systems.

Chapter 11

Open Challenges and Outlook

In this chapter, we present the main identified challenges of the proposed approach and how we started to tackle some of them. We then present the long-term vision for the proposed DPE approach for continuous and automated execution of performance tests alongside the CSDL.

11.1 Limitations

The main mentioned limitations by the participants of the expert review and the summative evaluation, refer to the complexity in modeling the workload, the need for support from IDEs to successfully specify tests due to the richness and expressiveness of the DSL, and to the need for another layer of abstraction to make the test specification even more human-readable. I completely agree with the feedback, and we got similar feedback while developing and applying the approach in multiple use-cases collaborating with many researchers. Some of the limitations have been already preliminary tackled in work we did in collaboration with other researchers, and their overcoming contributes to enabling our long-term vision of DPE approaches for performance testing automation and integration in CSDL. We present an approach tackling the complexity in modeling the workload as part of the case study discussed in Sect. 8.2, and an approach working on an additional layer of abstraction to increase human-readability of the specified tests in Sect. 8.3. The integration with IDEs is part of the future work and enabled by the provided DSL library, and the BenchFlow APIs.

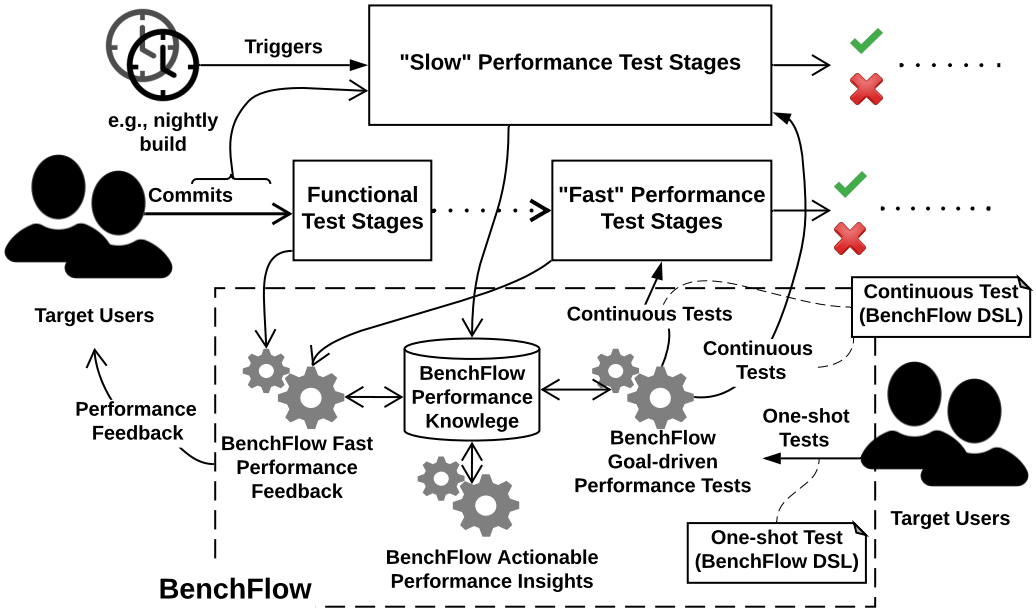

Figure 11.1. Long-term Vision and Outlook

11.2 Long-term Vision and Outlook

In Fig. 11.1 we present our long-term vision and outlook for DPE approaches for performance testing automation and integration in CSDL. The overall vision is realized by four main building blocks:

1) *CSDL Lifecycle Integration*: define suitable ways to enable the users to express the moment in the CSDL in which to **integrate** which performance tests/queries, limiting the impact on the "continuity" of the lifecycle and deciding about the result of CSDL processes according to the executed tests result. Some performance tests, relatively fast, mandatory to proceed to the next stages of development are executed continuously during active development. Some other performance tests, more complex and slow compared to the previous ones, are executed in different moments, for example overnight;

2) *BenchFlow Goal-driven Performance Tests*: enable users to declaratively express **goal-driven performance questions** [Walter et al., 2016], directly schedule the execution of the defined test or trigger them from CSDL processes, and obtain test results;

3) *BenchFlow Fast Performance Feedback*: provide **fast performance feedback** to the users relying on performance knowledge collected from the SUT, so that they can continuously gain performance knowledge about the SUT and use this knowledge to improve the same;

4) *BenchFlow Actionable Performance Insights*: develop suitable ways to analyze collected performance knowledge about the SUT, to provide users with **actionable insights** about the SUT's performance employing pre-analyzed performance reports ad automatically interpreted performance results determined by the SUT type and deployment environment, useful to improve the SUT's quality.

In this work we provide solutions towards the first two building blocks, being the integration in the CSDL lifecycle (Sect. 4.4), for which we also contribute an extension in a join research work with other researchers presented in Sect. 8.2, and the support for declarative performance test specifications (Chap. 5) and execution (Chap. 6) based on test goals for a subset of tests identified in a proposed automation-oriented performance tests catalog (Chap. 4).
We identify additional work to be conducted in integrating the proposed approach in IDEs and other tools used by developers so that they can easily define and validate declarative performance test specifications while developing the software. Other identified needed integration in the overall CSDL tools landscape important to facilitate shift-left of performance testing activities and proactive performance testing are also with tools such as APMs. Additionally, due to the increasing adoption of platforms based on Kubernetes [StackRox, 2020] for container orchestration, we foresee the need of extending the DSL for integrating additional Kubernetes capabilities in controlling and configuring SUT deployment, as well as BenchFlow services dedicated to automatic SUT deployment to accommodate the native integration with Kubernetes.
In providing our solutions, we also started to tackle the third building block, by enabling reuse of collected performance knowledge with a first and simplified approach and supporting the performance space exploration using predictive models. Additional work is required to tackle the third building block, which is the most challenging in terms of research and development, in our opinion. We identify the need of: a) modeling the performance of the SUT; b) enabling portability of performance models across different phases of the development [Lizhi et al., 2020] as well as across different systems and deployment environment [Yang et al., 2005; Zheng et al., 2015; Aleti et al., 2018]; c) dynamically determining tests to be executed in CSDL according to changes applied to the SUT and its development stage, and current knowledge about

the SUT performance to maximize the execution of tests likely to fail users' defined performance quality criteria [Schermann and Leitner, 2018; Hashemian et al., 2017; Mostafa et al., 2017; AlGhamdi et al., 2020; Mühlbauer et al., 2020; Gazzola et al., 2020; Schulz and van Hoorn, 2020]. In this context, joining the results of this dissertation with the results of the dissertation by [Walter, 2018] would contribute to the overall DPE vision by merging the declarative approach for performance test execution of this thesis, with the automated engine for performance engineering approach selection presented in [Walter, 2018]. The automated engine is selecting among different possible approaches available in performance engineering to answer the questions a user has, one of them being the execution of targeted performance tests.
Collaborating with other researchers we contribute initial work towards providing solutions for the fourth building block, as presented in Sect. 8.3. The overall long-term vision for the fourth building block includes the generation of performance test's results interpretation [Lloyd et al., 2014], providing insight to the users about the performance of the SUT, what is limiting its performance and how to improve the same to better match the performance quality criteria.
By relying on the approach proposed in this work, and extending it towards our long-term vision, we are confident DPE can become part of every day performance testing automation activities. We envision performance test execution is going to be integrated more and more in all the phases of CSDL, enabling different users and systems in defining and executing performance tests triggered according to events generated in CSDL by humans or by other systems. Resistance to change is going always to impact the adoption, but different strategies can be put in place to reduce the impact of this, for example, converted from standard and widely-used approaches to DPE ones and openness-oriented development to always enable other tools to integrate with tools built in the context of DPE.

11.3 Final Remarks

In this dissertation, we claim that a DPE approach allows for better integration of performance testing automation with CSDL when compared to standard imperative approaches, and provides means for different users to specify and/or execute performance testing activities during software development and release. We highlight the importance of relying on an expressive DSL for test specification and a model-driven framework, allowing the user to specify the test goal

and configure their automation in a declarative manner and schedule them for automated execution. This thesis points out the need for a new paradigm to look at performance testing automation activities, to favor performance test execution and shift-left of such activities that are historically executed in later stages of software development. Experience in different case studies, in many different domains, collaboration and feedback from other researchers in the area of performance engineering, and the results of the expert review and the summative evaluation we conducted hints us our claims find support in the community and motivate the conducted research and results, as well as the proposed long-term vision.

Part V

Appendix

Appendix A

DSL: Complete Test and Experiment YAML Specification

```
version: { String : <Test DSL version, e.g. '1'> }
name: { String : <name of the performance test> }
# OPTIONAL
description: { String : <description of the performance test> }
# OPTIONAL
labels: { [String] : <a list of labels> }
configuration:
  goal:
    type: { String : <"load" | "smoke" | "sanity" | "configuration" |
    ↪  "scalability" | "spike" | "exhaustive_exploration" | "stability_boundary"
    ↪  | "capacity_constraints" | "regression_complete" |
    ↪  "regression_intersection" | "acceptance"> }
    # OPTIONAL
    stored_knowledge: { Boolean : <"true" | "false"> } #default "false"
    observe:
      # OPTIONAL
      workloads:
        # OPTIONAL
        { String : <name of workload> }: { [String] : <a list of workload metrics>
        ↪  }
        # OPTIONAL
        { String : <name of workload.operation> }: { [String] : <a list of
        ↪  workload metrics> }
        ...
      # OPTIONAL
      services:
        { String : <name of service> }: { [String] : <a list of service metrics> }
        ...
```

```
    # OPTIONAL
    exploration:
      exploration_space:
        services:
          { String : <name of service> }:
              # OPTIONAL
              resources:
                cpu:
                  # EITHER specific values
                  values: { [String] : <a list of values + unit> }
                  # OR range
                  range: { [String, String] : <a list of values + unit
                  ↪  (inclusive)> }
                  # IF range we can specify step
                  step: { String : <step between the values in the range as a
                  ↪  mathematical expression: +,-,*,/,^><values + unit> }
                memory:
                  # EITHER specific values
                  values: { [String] : <a list of bytes value + unit> }
                  # OR range
                  range: { [String, String] : <a list of bytes value + unit
                  ↪  (inclusive)> }
                  # IF range we can specify step
                  step: { String : <step between the values in the range as a
                  ↪  mathematical expression: +,-,*,/,^><bytes value + unit> }
              # OPTIONAL
              configuration:
                { String : <name of environment variable> }: { [String] : <a list
                ↪  of possible values> }
                ...
          ...
        load_function:
          # OPTIONAL
          users: { [Number] : <a list of numbers> }
          # OPTIONAL
          users_range: { [Number, Number] : <a range of numbers (inclusive)> }
          # OPTIONAL
          users_step: { String : <step between the values in the range as a
          ↪  mathematical expression: +,-,*,/,^><number> }
        # OPTIONAL IF goal.type = stability_boundary
        stability_criteria:
          services:
            { String : <name of service> }:
              # OPTIONAL
              avg_cpu: { String : <a mathematical expression:
              ↪  >,<,>=,<=,=><number><"%"> }
```

```
            # OPTIONAL
            avg_memory: { String : <a mathematical expression:
             ↪  >,<,>=,<=,=><number><"%"> }
          ...
        workloads:
          { String : <name of workload> }:
          ...
    exploration_strategy:
      selection: { String : <"one_at_a_time" | "random_breakdown" |
       ↪  "stability_boundary_first"> }
      # OPTIONAL
      validation: { String : <random_validation_set> }
      # OPTIONAL
      regression: { String : <mars> }
load_function:
  # OPTIONAL IF SPECIFIED IN EXPLORATION
  users: { Number : <total number of users to be simulated> }
  ramp_up: { String : <amount of time><unit> }
  steady_state: { String : <amount of time><unit> }
  ramp_down: { String : <amount of time><unit> }
# OPTIONAL
termination_criteria:
  # OPTIONAL
  test:
    max_time: { String : <amount of time><unit> }
    # OPTIONAL
    max_number_of_experiments: { Number : <maximum number of experiments> }
    # OPTIONAL
    max_failed_experiments: { String : <number><"%"> }
  # OPTIONAL
  experiment:
    max_number_of_trials: { Number : <maximum number of trials> }
    # OPTIONAL
    max_failed_trials: { String : <number><"%"> }
    # OPTIONAL
    workloads:
      # OPTIONAL
      { String : <name of workload> }:
        confidence_interval_metric: { String : <a workload metrics> }
        confidence_interval_value: { Number : <value of the workload metric> }
        confidence_interval_precision: { String : <number><"%"> }
      ...
    # OPTIONAL
    services:
      { String : <name of service> }:
        confidence_interval_metric: { String : <a service metrics> }
```

```
        confidence_interval_value: { Number : <value of the service metric> }
        confidence_interval_precision: { String : <number><"%"> }
      ...
# OPTIONAL
quality_gates:
  # OPTIONAL IF exploration_strategy.regression = mars
  mean_absolute_error: { String : <number><"%"> }
  # OPTIONAL IF goal.type = regression_complete OR goal.type =
   ↪  regression_intersection
  regression:
    # EITHER service name
    service: { String : <name of service> }
    # OR workload name
    workload: { String : <name of workload> }
    gate_metric: { String : <name of the workload or service metric> }
    # OPTIONAL
    regression_delta_absolute: { String : <amount><unit> }
    # OPTIONAL
    regression_delta_percent: { String : <number><"%"> }
  # OPTIONAL
  workloads:
    { String : <name of workload> }:
        # OPTIONAL
      - max_mix_deviation: { String : <number><"%"> }
        # OPTIONAL
        max_think_time_deviation: { String : <number><"%"> }
        # OPTIONAL
        gate_metric: { String : <name of the workload metric> }
        # OPTIONAL
        condition: { String : <a mathematical expression: >,<,>=,<=,=,+%,-%> }
        # OPTIONAL
        gate_threshold_target: { String : <a threshold value> OR Number : <value
         ↪  of the workload metric> }
        # OPTIONAL
        gate_threshold_minimum: { String : <a threshold value> OR Number :
         ↪  <value of the workload metric> }
      ...
    ...
  # OPTIONAL
  services:
    { String : <name of service> }:
      - gate_metric: { String : <name of the service metric> }
        condition: { String : <a mathematical expression: >,<,>=,<=,=,+%,-%> }
        gate_threshold_target: { String : <a threshold value> OR Number : <value
         ↪  of the service metric> }
        # OPTIONAL
```

```
          gate_threshold_minimum: { String : <a threshold value> OR Number :
          ↪  <value of the service metric> }
        ...
      ...
sut:
  name: { String : <name of the sut> }
  versions:
    # EITHER specific values
    values: { [String] : <name of versions in the wanted order> }
    # OR range
    range: { [String, String] : <a list of versions (inclusive)> }
  # OPTIONAL
  type: { String : <"wfms" | "http"> }
  sut_configuration:
    default_target_service:
      name: { String : <name of target service> }
      endpoint: { String : <base endpoint for each operation> }
      # OPTIONAL
      sut_ready_log_check: { String : <a regular expression to check the
      ↪  availability of the sut> }
    # OPTIONAL
    deployment:
      { String : <name of service> }: { String : <name of server alias> }
      ...
  # OPTIONAL
  services_configuration:
    { String: <name of service> }:
      # OPTIONAL
      resources:
        # OPTIONAL
        cpu: { [String] : <a list of values + unit> }
        # OPTIONAL
        memory: { [String] : <a list of bytes value + unit> }
      # OPTIONAL
      configuration:
        { String: <name of environment variable> }: { String: <value> }
        ...
    ...
workloads:
  { String : <name of the workload> }
    # OPTIONAL IF more than one workload
    popularity: { String : <number><"%"> }
    { String : <name of the workload item> }
      driver_type: { String : <"start_bpmn" | "http"> }
      # OPTIONAL IF goal.type = regression_complete OR goal.type =
      ↪  regression_intersection
```

```
192       sut_version: { String : <sut version> }
193       # OPTIONAL
194       target_service: { String : <name of service> } # default to
          ↪ default_target_service
195       # OPTIONAL
196       inter_operation_timings:  { String : <"negative_exponential" | "uniform" |
          ↪ "fixed_time"> }
197       # OPTIONAL IF more than one workload item
198       popularity: { String : <number><"%"> }
199       # IF driver_type = start_bpmn (WfMS)
200       operations:
201         - { String : <name of the .bpmn file with the process> }
202         ...
203       # IF driver_type = http (Web Services)
204       operations:
205         { String : <Name of operation> }
206           protocol: { String : <"http" | "https"> }
207           endpoint: { String : <path to call optionally referencing to data in
              ↪ body, parameter or extracted from response to other operations> }
208           # REFERENCE:
              ↪ https://www.w3.org/Protocols/rfc2616/rfc2616-sec5.html#sec5.1.1
209           method: { String : <"OPTIONS" | "GET" | "HEAD" | "POST" | "PUT" |
              ↪ "DELETE" | "TRACE" | "CONNECT"> }
210           # OPTIONAL
211           # REFERENCE: https://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html
212           headers:
213             Accept: { String : <Media Type> }
214             ...
215           # OPTIONAL
216           query_parameter:
217             - { String : <name of the query parameter> }:
218               items:
219                 - { String : <content value> }
220                 ...
221               # OPTIONAL
222               retrieval: { String : <"sequential" | "random"> }
223             ...
224           # OPTIONAL
225           url_parameter:
226             - { String : <name of the url parameter> }:
227               items:
228                 - { String : <content value> }
229                 ...
230               # OPTIONAL
231               retrieval: { String : <"sequential" | "random"> }
232             ...
```

```
          # OPTIONAL
          extract_regex:
            - { String : <name assigned to the extracted value> }:
              # REFERENCE: https://developer.mozilla
               ↪  .org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/match
              pattern: { String : <a regex pattern to extract values> }
              # OPTIONAL
              default: { String : <the default value assigned when no values are
               ↪  extracted> }
              # OPTIONAL
              match_number: { Number : <the matched element index to select> }
            ...
          # OPTIONAL
          extract_json:
            - { String : <name assigned to the extracted value> }:
              # REFERENCE: https://goessner.net/articles/JsonPath/
              pattern: { String : <a JSON selector pattern to extract values> }
              # OPTIONAL
              default: { String : <the default value assigned when no values are
               ↪  extracted> }
              # OPTIONAL
              match_number: { Number : <the matched element index to select> }
            ...
          # OPTIONAL
          body:
            content:
              - { String : <content value> }
              ...
            # OPTIONAL
            retrieval: { String : <"sequential" | "random"> }
          # OPTIONAL
          body_file: { String : <name of a data_sources> }
          # OPTIONAL
          body_form:
            - { String : <name of the form field> }:
              items:
                - { String : <content value> }
                ...
              # OPTIONAL
              retrieval: { String : <"sequential" | "random"> }
            ...
      # OPTIONAL IF driver_type = http (Web Services)
      data_sources:
        - path: { String : <path to a file containing data> }
          delimiter: { String : <delimiter used in the file containing data> }
          # OPTIONAL
```

```
          name: { String : <name assigned to the data source> }
          # OPTIONAL
          retrieval: { String : <"sequential" | "random"> }
        ...
      # OPTIONAL
      mix:
        # IF FixedSequenceMix
        fixed_sequence: { [String] : <name of operations in the wanted order> }
        # IF FlatMix. List of percentages (SUM = 100%) and optionally think time
        ↪ (tt), index refers to index of operation
        flat: { [String] : <number><"%"> <tt({ Number : <mean> } { Number :
        ↪ <deviation> })> }
        # IF FlatSequenceMix. List of percentages (SUM = 100%) and optionally
        ↪ think time (tt), index refers to index of sequence specified below
        flat: { [String] : <number><"%"> <tt({ Number : <mean> } { Number :
        ↪ <deviation> })> }
          sequences:
            - { [String] : <name of operations in the wanted order> }
            ...
        # IF MatrixMix (needs to be a square matrix). List of percentages (SUM =
        ↪ 100%) and optionally think time (tt), index refers to index of
        ↪ operation
        matrix:
          - { [String] : <number><"%"> <tt({ Number : <mean> } { Number :
          ↪ <deviation> })> }
          ...
    ...
  ...
# OPTIONAL
data_collection:
  only_declared: { Boolean : <"true" | "false"> } # default "false"
  # OPTIONAL
  services:
    # IF collector does NOT require CONFIGURATION
    { String : <name of service> }: { String | [String] : <names of BenchFlow
    ↪ collectors> }
    # IF some collectors requires CONFIGURATION
    { String : <name of service> }:
      { String : <name of BenchFlow collector> }:
        configuration:
          { String : <name of environment variable> }: { [String] : <a list of
          ↪ possible values> }
          ...
      ...
    ...
  # OPTIONAL
```

```
313   workloads:
314     { String : <name of the workload> }:
315       # OPTIONAL IF trial execution framework = JMeter
316       jmeter:
317         interval: { String : <time interval for data collection><"s" (seconds)> }
            ↪  # default 1s
318       # OPTIONAL IF trial execution framework = Faban
319       faban:
320         interval: { String : <time interval for data collection><"s" (seconds)> }
            ↪  # default 1s
321     ...
```

Listing A.1. DSL: Complete Test YAML Format Specification

```
1   version: { String : <Experiment DSL version, e.g. "1"> }
2   name: { String : <name of the performance experiment derived from the name of the
      ↪  test> }
3   # OPTIONAL
4   description: { String : <description of the performance experiment derived from
      ↪  the name of the test> }
5   # OPTIONAL
6   labels: { [String] : <a list of labels> }
7   configuration:
8     # SET dependent on the load_function or exploration specification at test level
9     load_function:
10      users: { Number : <total number of users to be simulated> }
11      ramp_up: { String : <amount of time><unit> }
12      steady_state: { String : <amount of time><unit> }
13      ramp_down: { String : <amount of time><unit> }
14    termination_criteria:
15      max_time: { String : <amount of time><unit> }
16      # OPTIONAL
17      experiment:
18        max_number_of_trials: { Number : <maximum number of trials> }
19        # OPTIONAL
20        max_failed_trials: { String : <number><"%"> }
21        # OPTIONAL
22        workloads:
23          # OPTIONAL
24          { String : <name of workload> }:
25            confidence_interval_metric: { String : <a workload metrics> }
26            confidence_interval_value: { Number : <value of the workload metric> }
27            confidence_interval_precision: { String : <number><"%"> }
```

```
        ...
      # OPTIONAL
      services:
        { String : <name of service> }:
          confidence_interval_metric: { String : <a service metrics> }
          confidence_interval_value: { Number : <value of the service metric> }
          confidence_interval_precision: { String : <number><"%"> }
        ...
sut:
  name: { String : <name of the sut> }
  version: { String : <name of version> }
  # OPTIONAL
  type: { String : <"wfms" | "http"> }
  sut_configuration:
    default_target_service:
      name: { String : <name of target service> }
      endpoint: { String : <base endpoint for each operation> }
      # OPTIONAL
      sut_ready_log_check: { String : <a regular expression to check the
       ↪  availability of the sut> }
    # OPTIONAL
    deployment:
      { String : <name of service> }: { String : <name of server alias> }
      ...
  # OPTIONAL. SET dependent on the load_function or exploration specification at
   ↪  test level
  services_configuration:
    { String: <name of service> }:
      # OPTIONAL
      resources:
        # OPTIONAL
        cpu: { [String] : <a list of values + unit> }
        # OPTIONAL
        memory: { [String] : <a list of bytes value + unit> }
      # OPTIONAL
      configuration:
        { String: <name of environment variable> }: { String: <value> }
        ...
    ...
workloads:
  { String : <name of the workload> }
    # OPTIONAL IF more than one workload
    popularity: { String : <number><"%"> }
    { String : <name of the workload item> }
      driver_type: { String : <"start_bpmn" | "http"> }
      # OPTIONAL IF goal.type = regression_complete OR goal.type =
       ↪  regression_intersection
```

```
72      sut_version: { String : <sut version> }
73      # OPTIONAL
74      target_service: { String : <name of service> } # default to
        ↪ default_target_service
75      # OPTIONAL
76      inter_operation_timings:  { String : <"negative_exponential"  "uniform"
        ↪ "fixed_time"> }
77      # OPTIONAL IF more than one workload item
78      popularity: { String : <number><"%"> }
79      # IF driver_type = start_bpmn (WfMS)
80      operations:
81        - { String : <name of the .bpmn file with the process> }
82        ...
83      # IF driver_type = http (Web Services)
84      operations:
85        { String : <Name of operation> }
86          protocol: { String : <"http" | "https"> }
87          endpoint: { String : <path to call optionally referencing to data in
            ↪ body, parameter or extracted from response to other operations> }
88          # REFERENCE:
            ↪ https://www.w3.org/Protocols/rfc2616/rfc2616-sec5.html#sec5.1.1
89          method: { String : <"OPTIONS"  "GET"  "HEAD"  "POST"  "PUT"  "DELETE"
            ↪ "TRACE" | "CONNECT"> }
90          # OPTIONAL
91          # REFERENCE: https://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html
92          headers:
93            Accept: { String : <Media Type> }
94            ...
95          # OPTIONAL
96          query_parameter:
97            - { String : <name of the query parameter> }:
98              items:
99                - { String : <content value> }
100               ...
101             # OPTIONAL
102             retrieval: { String : <"sequential" | "random"> }
103           ...
104         # OPTIONAL
105         url_parameter:
106           - { String : <name of the url parameter> }:
107             items:
108               - { String : <content value> }
109               ...
110             # OPTIONAL
111             retrieval: { String : <"sequential" | "random"> }
112           ...
```

```
          # OPTIONAL
          extract_regex:
            - { String : <name assigned to the extracted value> }:
              # REFERENCE: https://developer.mozilla
               ↪  .org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/match
              pattern: { String : <a regex pattern to extract values> }
              # OPTIONAL
              default: { String : <the default value assigned when no values are
               ↪  extracted> }
              # OPTIONAL
              match_number: { Number : <the matched element index to select> }
            ...
          # OPTIONAL
          extract_json:
            - { String : <name assigned to the extracted value> }:
              # REFERENCE: https://goessner.net/articles/JsonPath/
              pattern: { String : <a JSON selector pattern to extract values> }
              # OPTIONAL
              default: { String : <the default value assigned when no values are
               ↪  extracted> }
              # OPTIONAL
              match_number: { Number : <the matched element index to select> }
            ...
          # OPTIONAL
          body:
            content:
              - { String : <content value> }
              ...
            # OPTIONAL
            retrieval: { String : <"sequential" | "random"> }
          # OPTIONAL
          body_file: { String : <name of a data_sources> }
          # OPTIONAL
          body_form:
            - { String : <name of the form field> }:
              items:
                - { String : <content value> }
                ...
              # OPTIONAL
              retrieval: { String : <"sequential" | "random"> }
            ...
      # OPTIONAL IF driver_type = http (Web Services)
      data_sources:
        - path: { String : <path to a file containing data> }
          delimiter: { String : <delimiter used in the file containing data> }
          # OPTIONAL
```

```
          name: { String : <name assigned to the data source> }
          # OPTIONAL
          retrieval: { String : <"sequential" | "random"> }
        ...
      # OPTIONAL
      mix:
        # IF FixedSequenceMix
        fixed_sequence: { [String] : <name of operations in the wanted order> }
        # IF FlatMix. List of percentages (SUM = 100%) and optionally think time
        ↪  (tt), index refers to index of operation
        flat: { [String] : <number><"%"> <tt({ Number : <mean> } { Number :
        ↪  <deviation> })> }
        # IF FlatSequenceMix. List of percentages (SUM = 100%) and optionally
        ↪  think time (tt), index refers to index of sequence specified below
        flat: { [String] : <number><"%"> <tt({ Number : <mean> } { Number :
        ↪  <deviation> })> }
          sequences:
            - { [String] : <name of operations in the wanted order> }
            ...
        # IF MatrixMix (needs to be a square matrix). List of percentages (SUM =
        ↪  100%) and optionally think time (tt), index refers to index of
        ↪  operation
        matrix:
          - { [String] : <number><"%"> <tt({ Number : <mean> } { Number :
          ↪  <deviation> })> }
          ...
    ...
  ...
# DEPENDENT on the observe and the data_collection sections in the test
↪  specification
data_collection:
  only_declared: { Boolean : <"true" | "false"> } # default "false"
  # OPTIONAL.
  services:
    # IF collector does NOT require CONFIGURATION
    { String : <name of service> }: { String | [String] : <names of BenchFlow
    ↪  collectors> }
    # IF some collectors requires CONFIGURATION
    { String : <name of service> }:
      { String : <name of BenchFlow collector> }:
        configuration:
          { String : <name of environment variable> }: { [String] : <a list of
          ↪  possible values> }
          ...
      ...
    ...
```

```
192   # OPTIONAL
193   workloads:
194     { String : <name of the workload> }:
195       # OPTIONAL IF trial execution framework = JMeter
196       jmeter:
197         interval: { String : <time interval for data collection><"s" (seconds)> }
            ↪  # default 1s
198       # OPTIONAL IF trial execution framework = Faban
199       faban:
200         interval: { String : <time interval for data collection><"s" (seconds)> }
            ↪  # default 1s
201     ...
```

Listing A.2. DSL: Complete Experiment YAML Format Specification

Appendix B

DSL: YAML Specification Examples

```yaml
version: "3"
name: "Predictive Stability Boundary Test"
description: "Example of Predictive Stability Boundary Test re-using Store Knowledge and Observing Client- and Server-side Metrics"
labels: "stability_boundary", "predictive", "stored_knowledge"
configuration:
  goal:
    type: "stability_boundary"
    stored_knowledge: "true"
    observe:
      workloads:
        workload_a: avg_response_time, avg_latency
        workload_b: avg_response_time, avg_latency
        workload_b.operation_a: avg_response_time
      services:
        service_a: avg_cpu, avg_memory
        service_b: avg_cpu, avg_memory
        dbms_a: avg_cpu, avg_memory, avg_io
    exploration:
      exploration_space:
        services:
          service_a:
            resources:
              cpu:
                range: [100m, 1000m]
                step: "*4"
              memory:
                range: [256Mi, 1024Mi]
```

```yaml
              step: "+768Mi"
          configuration:
            NUM_SERVICE_THREAD: [12, 24]
        dbms_a:
          resources:
            cpu:
              range: [100m, 1000m]
              step: "*10"
            memory:
              range: [256Mi, 1024Mi]
              step: "+768Mi"
          configuration:
            QUERY_CACHE_SIZE: 48Mi
      stability_criteria:
        services:
          service_a:
            avg_cpu: "<=60%"
            avg_memory: "<=80%"
          dbms_a:
            avg_memory: "<=70%"
        workloads:
          workload_b:
            max_mix_deviation: 5%
    exploration_strategy:
      selection: "stability_boundary_first"
      validation: "random_validation_set"
      regression: "mars"
load_function:
  users: 1000
  ramp_up: 5m
  steady_state: 20m
  ramp_down: 5m
termination_criteria:
  test:
    max_time: 120h
    max_failed_experiments: 10%
  experiment:
    max_failed_trials: 10%
    workloads:
      workload_a:
        confidence_interval_metric: avg_response_time
        confidence_interval_value: 50ms
        confidence_interval_precision: 95%
    services:
      service_a:
        confidence_interval_metric: avg_cpu
```

```yaml
          confidence_interval_value: 60%
          confidence_interval_precision: 95%
  quality_gates:
    mean_absolute_error: 10%
    workloads:
      workload_a:
        - max_mix_deviation: 5%
          max_think_time_deviation: 2%
          gate_metric: avg_response_time
          condition: "<="
          gate_threshold_target: "100ms"
          gate_threshold_minimum: "200ms"
    services:
      service_a:
        - gate_metric: avg_cpu
          condition: "<="
          gate_threshold_target: 50%
          gate_threshold_minimum: 60%
sut:
  name: "my_app"
  versions:
    values: "v1.5"
  type: "http"
  sut_configuration:
    default_target_service:
      name: "service_a"
      endpoint: "/"
      sut_ready_log_check: "/(.*)System started(.*)/g"
    deployment:
      service_a: "my_server"
  services_configuration:
    service_b:
      resources:
        cpu: 200m
        memory: 256Mi
      configuration:
        THREADPOOL_SIZE: 64
workloads:
  workload_a:
    popularity: 70%
    item_a:
      driver_type: "http"
      inter_operation_timings: "negative_exponential"
      popularity: 80%
      operations:
        operation_a:
```

```
          protocol: "https"
          endpoint: "/"
          method: "GET"
          extract_regex:
            - title:
              pattern: "<title>(.*?)</title>"
              default: ""
              match_number: 1
        operation_b:
          protocol: "https"
          endpoint: "/${title}"
          method: "POST"
          body_file: "datasource_a"
        data_sources:
          - path: "/path_to_datasource_a"
            delimiter: ","
            name: datasource_a
            retrieval: "random"
        mix:
          flat: "75.0% tt(1000.0 500.0), 25.0% tt(2000.0 400.0)"
    item_b:
      driver_type: "http"
      inter_operation_timings: "negative_exponential"
      popularity: 20%
      operations:
        operation_a:
          protocol: "https"
          endpoint: "/"
          method: "GET"
  workload_b:
    popularity: 30%
    item_a:
      driver_type: "http"
      target_service: "service_b"
      operations:
        operation_a:
          protocol: "https"
          endpoint: "/"
          method: "GET"
data_collection:
  # AUTOMATICALLY attached based on the observe section IF NOT specified
  services:
    service_a: "stats"
    service_b: "stats"
    dbms_a: "stats"
```

Listing B.1. DSL: A Predictive Stability Boundary Test YAML Example

Appendix C

Expert Review Survey

C.1 Declarative Performance Testing

The Proposed Approach for Declarative Performance Tests Specification and Execution
This approach is part of a PhD contribution. We ask you to contribute with your valuable feedback to complete the evaluation section of the same. The answers are going to be reported anonymously as part of the thesis. We value the time you spend on providing the feedback, so we want to give you back something to show our appreciation. We are giving away three little gifts as a sign of appreciation, raffled among all the participants completing the survey. More details on this at the end of the survey, where you can decide whether to opt-in or not.

NOTE 1: due to limitations of Google Forms, we can not use a larger font size. To increase the font-size please follow the advice on the following guide: https://www.computerhope.com/issues/ch000779.htm (Suggested: use the topmost Tip of the referenced guide). Images are not going to be increased guaranteeing the quality, thus we always offer a full-size image link.
NOTE 2: going back to the previous page during the survey, usually saves the already provided answers when coming back to the page containing the answer. If you think you might need to do this, we advise you to test the behavior with your browser to be sure about it, before completing multiple answers

C.2 Expert Review of The Proposed Approach

Context
Recent industry trends show increasing adoption of Development and Operations (DevOps) practices. Alongside the adoption of DevOps, performance testing continues to evolve to meet the growing demands of the modern enterprise and its need for automation. Modeling and automated execution of performance tests are time-consuming and difficult activities, requiring expert knowledge, complex infrastructure, and a rigorous process to guarantee the quality of collected performance data and the obtained results. Currently available performance testing approaches are not well integrated with DevOps practices and tools and are often focusing only on specific needs of performance testing modeling and automation.

Proposal
We propose a **Declarative Approach for Performance Tests Execution**, enabling the continuous and automated execution of performance tests alongside the Continuous Software Development Lifecycle (CSDL)*.
The approach is comprised of a declarative Domain Specific Language (DSL) enabling the declarative specification of performance tests and their automated orchestration processes alongside the CSDL, and a framework for end-to-end automated performance testing relying on the contributed DSL. The main target systems are RESTful Web services and Business Process Model and Notation 2.0 (BPMN20) Workflow Management Systems (WfMSs) deployed using a deployment descriptor relying on Docker containers (More details on https://docs.docker.com/compose/compose-file/).
The main target users of our approach are developers, software testers, quality assurance engineers, and operations engineers. The users of our approach can specify performance tests using a goal-oriented approach, and declaratively configure the entire process followed to reach the goal and evaluate the test result using the same DSL. With the same DSL, they can build test suites and define when different tests are executed during the CSDL, according to specific triggers. The DSL is used to configure a framework programmed using such DSL and implementing the processes needed for executing performance experiments part of the declared test, and a control loop watching the state of the executing test and scheduling experiments to reach the specification stated in the declarative specification submitted by the users.
More details are provided in a dedicated section of this survey.

Expert Review Objective
The objective of this expert review is to collect feedback about the proposed approach in terms of **expressiveness** and **reusability** of the DSL, as well as perceived **usability** and **effort** of the same and **suitability** for the target users, especially comparing it to standard imperative approaches+, like the one proposed by JMeter, and considering the target users of the approach^.

Expert Review Target Audience
We mainly target performance engineering practitioners, both in academia and industry.

Structure of the Survey
The survey starts with a section where we ask you to fill in some information about your experience and expertise. We then present you the approach in a dedicated section where we go over all the elements of the proposed DSL, and we provide you with examples illustrating how a user is expected to state goal-driven declarative performance tests using our approach. After an introduction to the approach, we are going to ask you five multiple-choice questions about the same where you can also optionally provide additional feedback. Finally, we propose some questions to collect PROs and CONs about the approach and references to similar approaches.

Expected average completion time: up to 35-45 minutes, of which up to 20-30 minutes for learning about the proposed approach, depending on your background.

The survey is anonymous and no contact details are collected, unless provided by you at the end of the survey. Constructive feedback is always the most valuable.
* The approach has been first presented at the International Workshop on Quality-Aware DevOps in 2017 (paper: https://dl.acm.org/doi/pdf/10.1145/3053600.3053636, presentation: https://bit.ly/2Vtcqny) and then at the International Conference on Performance Engineering in 2018 (paper: https://dl.acm.org/doi/pdf/10.1145/3184407.3184417, presentation: https://bit.ly/3g7T7YP). It has been largely extended over time.
+ A brief overview of imperative approaches vs declarative approaches for testing can be found on https://wiki.saucelabs.com/display/DOCS/Best+Practice%3A+Imperative+v.+Declarative+Testing+Scenarios

^ Evaluating the framework allowing the automated execution of the declaratively specified performance tests is out of the scope for this survey since it is a research prototype and it is not widely used as other frameworks for performance testing.

C.3 Experience and Expertise

In this section, we are collecting some information about you, to better analyze the answers you are going to provide in the following sections.

C.3.1 Education

1. Latest degree of education

- ☐ Middle school.
- ☐ High School.
- ☐ Bachelor Degree.
- ☐ Master Degree.
- ☐ PhD.
- ☐ Other.

C.3.2 Professional Role

2. Your current professional role

- ☐ Software Developer.
- ☐ Software Engineer.
- ☐ Software Tester.
- ☐ DevOps Engineer.
- ☐ QA Analyst.
- ☐ Performance Analyst.
- ☐ Site Reliability Engineer.
- ☐ Researcher.
- ☐ Professor.
- ☐ Student.
- ☐ Other: _____.

C.3.3 Experience in the current professional role

3. Years of experience in the current professional role: _____

C.3.4 Experience related to software performance testing

4. Years of experience in the area of software performance testing. Not necessarily as the main activity. NOTE: it can be 0. : _____

C.3.5 Which are your main tasks and activities?

5. Describe your main tasks and activities in your current role: _____

C.3.6 How familiar are you with the following concepts?

State how familiar are you with the following concepts related to performance test implementation and execution in the context of this survey

6a. Performance Testing Not at all ☐—☐—☐—☐—☐ Expert

6b. System Under Test Not at all ☐—☐—☐—☐—☐ Expert

6c. System Workload Not at all ☐—☐—☐—☐—☐ Expert

6d. Performance Data Collection Not at all ☐—☐—☐—☐—☐ Expert

6e. System Performance Metrics Not at all ☐—☐—☐—☐—☐ Expert

6f. Performance Tests Execution Framework Not at all ☐—☐—☐—☐—☐ Expert

6g. Load Driver Not at all ☐—☐—☐—☐—☐ Expert

6h. Imperative Approaches for Performance Engineering (e.g., JMeter) Not at all ☐—☐—☐—☐—☐ Expert

6i. Declarative Performance Engineering Not at all ☐—☐—☐—☐—☐ Expert

6j. System Deployment Descriptor Not at all ☐—☐—☐—☐—☐ Expert

6k. Docker Container Not at all ☐—☐—☐—☐—☐ Expert

6l. Container Orchestrator Framework (e.g., Kubernetes) Not at all ☐—☐—☐—☐—☐ Expert

6m. Continuous Integration and Delivery Not at all □—□—□—□—□ Expert

6n. RESTful Web Services Not at all □—□—□—□—□ Expert

6o. YAML Language Not at all □—□—□—□—□ Expert

C.4 Overview of the Approach and Examples

In the following, we present an overview of the proposed approach. We illustrate the YAML specification format the users of our approach are expected to master for declaratively specifying performance tests and configuring the test's automation process. We then present some examples of declarative performance test specifications relying on the proposed YAML specification format. What users are expected to state for declarative performance test automation To rely on the proposed declarative performance test automation solution, users need to have a basic understanding of the performance testing domain* and are expected to define a test using the proposed YAML specification format and specify:

1) The test goal and its configuration, as well as the performance space to be explored if required by the goal (`goal`, `exploration_space`)

2) The load function (`load_function`)

3) The performance metrics of interest (`observe`)

4) The criteria to determine if the test execution has to be terminated before its completion (`termination_criteria`)

5) The expected outcome of the test according to defined quality criteria (`quality_gates`)

6) The workloads used for the test and how to mix its operations (`workloads`, `operations`, `mix`)

7) The system under test type, and its configuration (`sut`)

8) The data collection services needed to collect performance data useful to compute the metrics of interest (`data_collection`)

After specifying different tests, users can also define test suites to integrate subsets of tests as part of the CSDL. To do so, users are expected to define:

1) The test suite, selecting tests by referring to YAML files defining them or by selecting them using labels (`suite`, `tests`)

2) The triggers activating the test execution, according to CSDL events (`triggers`)

3) Optionally, the environments on which tests have to be executed (`environments`)

4) The expected outcome of the test suite according to defined quality criteria (`quality_gates`)

Users are expected to provide the test specifications, as well as test data and a system under test (SUT) deployment descriptor as part of a so-called *test bundle*. The SUT deployment descriptor defines the services part of the SUT, their configuration, deployment specification, and how they communicate among them. More details are provided on the official reference: https://docs.docker.com/compose/compose-file/.
The proposed framework provides a command-line interface (CLI) that can also be integrated into different Continuous Integration and Continuous Delivery pipelines, defined with tools such as Jenkins (https://www.jenkins.io/). Through such CLI the user can automatically submit tests and test suite specifications for execution to the proposed framework.
Test specifications can be reused, extended and overridden, to facilitate reuse of base test definitions for more complex tests. Some examples are provided in the examples section, presented after the YAML specification format.
In the following, we present the YAML specification format as well as examples of declarative performance tests specification and integration with CSDL.

* Note that this statement refers to performance test specification. For what concerns analyzing the performance results, and improving the system performance basic understanding of the performance domain might not be sufficient.

C.4.1 YAML Specification Format

Defining a Declarative Performance Test
Overall Test Specification

```
version: { String : <Test DSL version, e.g. '1'> }
name: { String : <name of the performance test> }
# OPTIONAL
description: { String : <description of the performance test> }
# OPTIONAL
labels: { [String] : <a list of labels> }
configuration:
  goal:
    ...
  load_function:
    # OPTIONAL IF SPECIFIED IN EXPLORATION
    users: { Number : <total number of users to be simulated> }
    ramp_up: { String : <amount of time><unit> }
    steady_state: { String : <amount of time><unit> }
    ramp_down: { String : <amount of time><unit> }
  # OPTIONAL
  termination_criteria:
    ...
  # OPTIONAL
  quality_gates:
    ...
sut:
  ...
workloads:
  ...
# OPTIONAL
data_collection:
  ...
```

Listing C.1. Expert Review: The Test YAML Format Specification

In Listing C.1, we report the specification of the overall test structure. As you can see, the user can specify all the information described above. According to the selected goal, the user is requested to configure the automation process followed to reach such goals.
For what concerns the `load_function`, the user can specify the wanted number of simulated users and characteristics of the load function in terms of time to go from zero to the number of specified users (`ramp_up`), time to issue the load to the system under test at the specified number of users (`steady_state`), and time to go from the wanted number of users back to zero (`ramp_down`).*
Among the other data, the user can also specify `labels` (line #6). Labels are important for selecting different test specifications when integrating test

execution in CSDL.
Full-size image: https://docs.google.com/drawings/d/1a9CSMBnDm8ToNRbKpj-Vb7Q7oxVR71T0V4E3ZBC1ba4

* We currently only support the mentioned shape of the load function, due to limitations in the frameworks we use to automate the performance test execution. Other frameworks can be integrated, thus the shapes of supported load functions could be enhanced.

The Goal Specification

```
1  configuration:
2   goal:
3     type: { String : < "load" | "smoke" | "sanity" | "configuration" |
      ↪  "scalability" | "spike" | "exhaustive_exploration" | "stability_boundary" |
      ↪  "capacity_constraints" | "regression_complete" | "regression_intersection"
      ↪  | "acceptance" > }
4     # OPTIONAL
5     stored_knowledge: { Boolean : <"true" | "false"> } # default "false"
6     observe:
7       ...
8     # OPTIONAL
9     exploration:
10      ...
```

Listing C.2. Expert Review: The Goal YAML Format Specification

In Listing C.2, we report the specification of the performance test goal. As you can see, the user can specify many different supported goals (line #3), such as `load test`, `smoke test`, `configuration test`, and `regression test`. The goal is usually mapped to well-known performance test types. The number of supported goals can be extended over time.
Among the other data, the user can also specify whether she wants to use `stored_knowledge` (line #4) or not. Reusing stored knowledge enables the possibility to reuse previous results of tests and experiments defining the exact same specification, executed as part of previous test execution, or as part of other tests specifications scheduled in parallel to the submitted one.
Full-size image: https://docs.google.com/drawings/d/1aEFxdmRDOFVeRC0OZRznZj-1saAX_SoVvDKuCTFkV6Y

* The actual values for the type of test can vary. Usually, also the variant of the name with the addition of the suffix "_test" is supported.

The Exploration Space Specification

```
goal:
  # OPTIONAL
  exploration:
    exploration_space:
      services:
        { String : <name of service> }:
            # OPTIONAL
            resources:
              cpu:
                # EITHER specific values
                values: { [String] : <a list of values + unit> }
                # OR range
                range: { [String, String] : <a list of values + unit (inclusive)>
                 ↪  }
                # IF range we can specify step
                step: { String : <step between the values in the range as a
                 ↪  mathematical expression: +,-,*,/,^><values + unit> }
              memory:
                # EITHER specific values
                values: { [String] : <a list of bytes value + unit> }
                # OR range
                range: { [String, String] : <a list of bytes value + unit
                 ↪  (inclusive)> }
                # IF range we can specify step
                step: { String : <step between the values in the range as a
                 ↪  mathematical expression: +,-,*,/,^><bytes value + unit> }
            # OPTIONAL
            configuration:
              { String : <name of environment variable> }: { [String] : <a list of
               ↪  possible values> }
              ...
        ...
      load_function:
        ...
      # OPTIONAL IF goal.type = stability_boundary
      stability_criteria:
        services:
          { String : <name of service> }:
```

```
34            # OPTIONAL
35            avg_cpu: { String : <a mathematical expression:
               ↪  >,<,>=,<=,=><number><"%"> }
36            # OPTIONAL
37            avg_memory: { String : <a mathematical expression:
               ↪  >,<,>=,<=,=><number><"%"> }
38          ...
39        workloads:
40          { String : <name of workload> }:
41          ...
42    exploration_strategy:
43      selection: { String : <"one_at_a_time" | "random_breakdown" |
         ↪  "stability_boundary_first"> }
44      # OPTIONAL
45      validation: { String : <random_validation_set> }
46      # OPTIONAL
47      regression: { String : <mars> }
```

Listing C.3. Expert Review: The Exploration Space and Exploration Strategy YAML Format Specification

In Listing C.3, we report the specification of the exploration space required by some of the supported goals, such as the `configuration test`.
When a complex test is specified, the user can define the exploration space in terms of `services' resources`, and `configuration` (a map of key-value pairs), and the `load function`. Services refer to services defined in the system under test deployment descriptor, provided as part of the test bundle.
For some goals, specific data have to be specified, such as for the "`stability_boundary`" one. In such cases, specific sections of the exploration space are provided. For the stability boundary test, the `stability criteria`, section is provided (lines #36-#47), allowing the user to specify stability criteria on services and workloads.
The exploration space could vary in size, thus the user can rely on a specification using a `range`, (e.g., line #13) of values to explore and a `mathematical function`, (e.g, line #15) to navigate the specified range.
Our approach computes the entire performance space to explore calculating the combinations without repetitions on all the possible values specified by the user in the configuration space, considering each specified dimension as a vector. For example, if the user defines an exploration space with a "service_a" having as resources value for memory: [128Mi, 256Mi] and a load function defining

100 as the number of users, then the exploration space is comprised of two experiments, one with 100 users and 128Mi of memory for "service_a", and a second one of with 100 users and 256Mi of memory for "service_a".
Given that the exploration space could explode in terms of experiments to be executed, we provide different strategies to explore the space (lines #48-#53). We also alert the user if the size of the space is greater than a threshold (default: 10)*. One of the possible strategies that can be specified, is a predictive strategy (lines #51-#53) attempting to approximate the performance of the SUT in the performance, space, thus reducing the number of executed experiments.
Full-size image: https://docs.google.com/drawings/d/1ah5nazm99zRtgJmb-JXNbyLkHc7VAHHwinJ9iSYrluc

* Users of the framework executing the specified test can specify the threashold as part of the framework configurations.

The Specification for the Performance Metrics to Observe

```
goal:
  observe:
   # OPTIONAL
   workloads:
     # OPTIONAL
     { String : <name of workload> }: { [String] : <a list of workload metrics> }
     # OPTIONAL
     { String : <name of workload.operation> }: { [String] : <a list of workload
     ↪  metrics> }
     ...
   # OPTIONAL
   services:
     { String : <name of service> }: { [String] : <a list of service metrics> }
     ...
```

Listing C.4. Expert Review: The Observe YAML Format Specification

In Listing C.4, we report the specification of the performance metrics of interest for the performance test. The user can request to `observe` a wide range of performance metrics and statistics provided by the proposed framework and computed on collected performance data, by referring to them by their names.

Performance metrics can be observed of both the `workloads`, representing client-side performance metrics, as well as for the `services`, representing server-side performance metrics. Examples of workload metrics are average response time and average latency. Examples of service metrics are average CPU utilization and maximum memory utilization. We also provide SUT-specific metrics, for example for BPMN20 WfMSs we provide custom metrics about number and time of executed process instances, as well as other ones (for more details, refer to section 4 of http://www.pautasso.info/biblio-pdf/benchflow-bpmds2017.pdf)
Full-size image: https://docs.google.com/drawings/d/1T2nd0I9EpDSoM3gVRV0n1BYDwHrHusNv8F8HOLaUZjI

The Termination Criteria Specification

```
configuration:
 # OPTIONAL
 termination_criteria:
   # OPTIONAL
   test:
     max_time: { String : <amount of time><unit> }
     # OPTIONAL
     max_number_of_experiments: { Number : <maximum number of experiments> }
     # OPTIONAL
     max_failed_experiments: { String : <number><"%"> }
   # OPTIONAL
   experiment:
     max_number_of_trials: { Number : <maximum number of trials> }
     # OPTIONAL
     max_failed_trials: { String : <number><"%"> }
     # OPTIONAL
     workloads:
       # OPTIONAL
       { String : <name of workload> }:
         confidence_interval_metric: { String : <a workload metrics> }
         confidence_interval_value: { Number : <value of the workload metric> }
         confidence_interval_precision: { String : <number><"%"> }
       ...
     # OPTIONAL
     services:
       { String : <name of service> }:
         confidence_interval_metric: { String : <a service metrics> }
         confidence_interval_value: { Number : <value of the service metric> }
         confidence_interval_precision: { String : <number><"%"> }
```

```
30      ...
```

Listing C.5. Expert Review: The Terminationa Criteria YAML Format Specification

In Listing C.5, we report the specification of the criteria used to determine if the test execution has to be terminated before its completion. Given performance tests may need a considerable amount of time to be executed, the support of tests exploring potentially vast performance space, and the intrinsic fact performance test can fail or have non-deterministic issues during execution (e.g., due to instability of the SUT, the load drivers, or the performance test execution framework), it is of utmost importance to specify criteria controlling when to stop tests that are expected not to be successful.
A termination criterion can be specified on the `test` itself (lines #5-#10), or on `experiments` (lines #12-#30). At the test level, the users can state the maximum time the test is allowed to be executed*, as well as upper bounds for the number of executed experiments or the number of failed experiments. At the experiment level, the user can specify the maximum number of trials (e.g., the number of repetitions a single experiment is allowed to perform to provide reliable performance metrics) as well as the maximum number of failing trials. She can then define termination criteria based on confidence intervals of workloads and services metrics. The confidence interval is then used to dynamically compute the number of trials+.
Termination criteria are evaluated during a test and experiment execution. The result is determined by evaluating every single criterion for the given entity and then applying the OR operator to the criterion conditions result. As soon as one criterion results verified for the given entity, the execution is terminated. For example, for a time-based test termination criterion, as soon as the criterion is met the test is terminated. For termination criterion based on confidence intervals for experiments, as soon as the criterion results valid, the iteration of the experiment in multiple trials is terminated.
Full-size image: https://docs.google.com/drawings/d/16uJHecrkh9ML_vUrFlR1b_YwJ-1V43yE1pJNss-Ji7w

* the proposed frameworks checks for consistency of the termination criteria, with the number and time needed for executing the experiments part of the test and warns the user in case of inconsistencies, e.g., on the total time

a test is allowed to execute
^ software systems metrics are expected to vary across multiple experiment iterations, thus it is important to repeat the experiment multiple times, to compute reliable performance metrics. More information can be found in section 4 of http://www.pautasso.info/biblio-pdf/benchflow-bpmds2017.pdf
+ More details about confidence intervals can be found on https://en.wikipedia.org/wiki/Confidence_interval and the referenced book: "A modern introduction to probability and statistics: understanding why and how"

The Quality Gates Specification

```
configuration:
 # OPTIONAL
 quality_gates:
   # OPTIONAL IF exploration_strategy.regression = mars
   mean_absolute_error: { String : <number><"%"> }
   # OPTIONAL IF goal.type = regression_complete OR goal.type =
    ↪ regression_intersection
   regression:
     # EITHER service name
     service: { String : <name of service> }
     # OR workload name
     workload: { String : <name of workload> }
     gate_metric: { String : <name of the workload or service metric> }
     # OPTIONAL
     regression_delta_absolute: { String : <amount><unit> }
     # OPTIONAL
     regression_delta_percent: { String : <number><"%"> }
   # OPTIONAL
   workloads:
     { String : <name of workload> }:
         # OPTIONAL
       - max_mix_deviation: { String : <number><"%"> }
         # OPTIONAL
         max_think_time_deviation: { String : <number><"%"> }
         # OPTIONAL
         gate_metric: { String : <name of the workload metric> }
         # OPTIONAL
         condition: { String : <a mathematical expression: >,<,>=,<=,=,+%,-%> }
         # OPTIONAL
         gate_threshold_target: { String : <a threshold value> OR Number : <value
          ↪ of the workload metric> }
         # OPTIONAL
```

```
31          gate_threshold_minimum: { String : <a threshold value> OR Number : <value
             ↪  of the workload metric> }
32        ...
33      ...
34    # OPTIONAL
35    services:
36      { String : <name of service> }:
37        - gate_metric: { String : <name of the service metric> }
38          condition: { String : <a mathematical expression: >,<,>=,<=,=,+%,-%> }
39          gate_threshold_target: { String : <a threshold value> OR Number : <value
             ↪  of the service metric> }
40          # OPTIONAL
41          gate_threshold_minimum: { String : <a threshold value> OR Number : <value
             ↪  of the service metric> }
42        ...
43      ...
```

Listing C.6. Expert Review: The Quality Gates YAML Format Specification

In Listing C.6, we report the specification of the `quality gates`. Users specify quality gates to define the outcome of the test according to different criteria, thus contributing to the goal-driven specification of the performance test. The outcome of the test at the end of its execution is usually `successful` or `failing`. The result is determined by evaluating every single gate and then applying the AND operator to the gates' results.
Quality gates can be specified on workloads (lines #18-#33) and service metrics (lines #35-#43). Gates are specified using dedicated metrics, such as `max_mix_deviation` (line #21), defining the maximum accepted deviation from the specified workload to interact with the SUT*, or `max_think_time_deviation` (line #23), defining the maximum accepted deviation in the time interleaving multiple operations interacting with the SUT with respect to the specified settings*. Conditions can also be specified on specific metrics (lines #24-#31 and #38-#41). For metrics, users can always report both a threshold indicating the wanted value for the metric, as well as a minimum accepted value. The gates are evaluated according to the minimum accepted value. Reporting the expected threshold as well is important for better evaluating the test result (more details on https://www.stickyminds.com/article/better-way-reporting-performance-test-results). When relying on an exploration strategy building a prediction model, the user can also specify a quality gate to determine when the test can be considered successful according

to the precision reached by the model (line #5).
For specific test types, such as the `regression`, is requested to specify customized quality gates. In this case, a specific section is provided as part of the language (lines #7-#16). We currently support a single regression condition based on an observed metric, applied either on a workload or a service. The condition checks for an absolute or a percentage variation among the versions specified as part of the regression test.
Full-size image: https://docs.google.com/drawings/d/13OIry1aP8t4FMqj8lbhFgamBkJMTU84VM4IeEik0_hI

* Refer to the next section for more details.

The Workloads, Operations, and their Mix Specification

```
1  workloads:
2   { String : <name of the workload> }
3     # OPTIONAL IF more than one workload
4     popularity: { String : <number><"%"> }
5     { String : <name of the workload item> }
6       driver_type: { String : <"start_bpmn" | "http"> }
7       # OPTIONAL IF goal.type = regression_complete OR goal.type =
        ↪ regression_intersection
8       sut_version: { String : <sut version> }
9       # OPTIONAL
10      target_service: { String : <name of service> } # default to
        ↪ default_target_service
11      # OPTIONAL
12      inter_operation_timings:  { String : <"negative_exponential" | "uniform" |
        ↪ "fixed_time"> }
13      # OPTIONAL IF more than one workload item
14      popularity: { String : <number><"%"> }
15      operations:
16        ...
17      # OPTIONAL IF driver_type = http (Web Services)
18      data_sources:
19        ...
20      # OPTIONAL
21      mix:
22        ...
23    ...
24  ...
```

Listing C.7. Expert Review: The Workload YAML Format Specification

```
workloads:
 { String : <name of the workload> }
   # OPTIONAL IF more than one workload
   popularity: { String : <number><"%"> }
   { String : <name of the workload item> }
     # IF driver_type = start_bpmn (WfMS)
     operations:
       - { String : <name of the .bpmn file with the process> }
       ...
     # IF driver_type = http (Web Services)
     operations:
       { String : <Name of operation> }
         protocol: { String : <"http" | "https"> }
         endpoint: { String : <path to call optionally referencing to data in
          ↪ body, parameter or extracted from response to other operations> }
         # REFERENCE:
          ↪ https://www.w3.org/Protocols/rfc2616/rfc2616-sec5.html#sec5.1.1
         method: { String : <"OPTIONS" | "GET" | "HEAD" | "POST" | "PUT" |
          ↪ "DELETE" | "TRACE" | "CONNECT"> }
         # OPTIONAL
         # REFERENCE: https://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html
         headers:
           Accept: { String : <Media Type> }
           ...
         # OPTIONAL
         query_parameter:
           - { String : <name of the query parameter> }:
             items:
               - { String : <content value> }
               ...
             # OPTIONAL
             retrieval: { String : <"sequential" | "random"> }
           ...
         # OPTIONAL
         url_parameter:
           - { String : <name of the url parameter> }:
             items:
               - { String : <content value> }
               ...
             # OPTIONAL
             retrieval: { String : <"sequential" | "random"> }
           ...
```

```
        # OPTIONAL
        extract_regex:
          - { String : <name assigned to the extracted value> }:
            # REFERENCE: https://developer.mozilla
            ↪ .org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/match
            pattern: { String : <a regex pattern to extract values> }
            # OPTIONAL
            default: { String : <the default value assigned when no values are
            ↪ extracted> }
            # OPTIONAL
            match_number: { Number : <the matched element index to select> }
          ...
        # OPTIONAL
        extract_json:
          - { String : <name assigned to the extracted value> }:
            # REFERENCE: https://goessner.net/articles/JsonPath/
            pattern: { String : <a JSON selector pattern to extract values> }
            # OPTIONAL
            default: { String : <the default value assigned when no values are
            ↪ extracted> }
            # OPTIONAL
            match_number: { Number : <the matched element index to select> }
          ...
        # OPTIONAL
        body:
          content:
            - { String : <content value> }
            ...
          # OPTIONAL
          retrieval: { String : <"sequential" | "random"> }
        # OPTIONAL
        body_file: { String : <name of a data_sources> }
        # OPTIONAL
        body_form:
          - { String : <name of the form field> }:
            items:
              - { String : <content value> }
              ...
            # OPTIONAL
            retrieval: { String : <"sequential" | "random"> }
          ...
    # OPTIONAL IF driver_type = http (Web Services)
    data_sources:
      - path: { String : <path to a file containing data> }
        delimiter: { String : <delimiter used in the file containing data> }
        # OPTIONAL
```

```
        name: { String : <name assigned to the data source> }
        # OPTIONAL
        retrieval: { String : <"sequential" | "random"> }
      ...
    # OPTIONAL
    mix:
      ...
  ...
...
```

Listing C.8. Expert Review: The Operations YAML Format Specification

```
workloads:
 { String : <name of the workload> }
   # OPTIONAL IF more than one workload
   popularity: { String : <number><"%"> }
   { String : <name of the workload item> }
     # OPTIONAL
     mix:
       # IF FixedSequenceMix
       fixed_sequence: { [String] : <name of operations in the wanted order> }
       # IF FlatMix. List of percentages (SUM = 100%) and optionally think time
       ↪  (tt), index refers to index of operation
       flat: { [String] : <number><"%"> <tt({ Number : <mean> } { Number :
       ↪  <deviation> })> }
       # IF FlatSequenceMix. List of percentages (SUM = 100%) and optionally think
       ↪  time (tt), index refers to index of sequence specified below
       flat: { [String] : <number><"%"> <tt({ Number : <mean> } { Number :
       ↪  <deviation> })> }
         sequences:
           - { [String] : <name of operations in the wanted order> }
           ...
       # IF MatrixMix (needs to be a square matrix). List of percentages (SUM =
       ↪  100%) and optionally think time (tt), index refers to index of
       ↪  operation
       matrix:
         - { [String] : <number><"%"> <tt({ Number : <mean> } { Number :
         ↪  <deviation> })> }
         ...
   ...
...
```

Listing C.9. Expert Review: The Mix YAML Format Specification

In Listing C.7, Listing C.8, and Listing C.9, we report the specification of the `workloads`, the `operations`, and their `mix` respectively. We support workload specifications for both RESTful Web services (selecting `http` as `driver_type` on line #4 of the workloads specification) and BPMN20 WfMSs (selecting `start_bpmn` as `driver_type` on line #4 of the workloads specification).
The user can specify multiple `workloads`, representing different interaction scenarios with the SUT. The workload can be realized by different workload items (e.g., representing different scenarios of interaction with the SUT), target a specific SUT version (line #6 of the workloads specification), and can specify the target service (line #8 of the workloads specification). It is also possible to specify how to select the timings among the workload items operations (line #10 of the workloads specification)* and the `popularity` (line #12 of the workloads specification) of each workload item, useful to decide the weight when multiple workload items are specified.
As part of the workload, the user specifies `operations`. According to the `driver_type` the user has different ways to specify the operations (lines #4-#75 of the operations specification). Operations can extract data from SUT's response to be used in subsequent operations (lines #38-#57 of the operations specification) and can refer to `data_source` (lines #76-#84 of the operations specification), as a source of data for calls to the SUT (lines #20-#37 and #58-#75 of the operations specification). We omit some details of operation specifications because they basically refer to the HTTP standard. For more information, you can refer to https://www.w3.org/Protocols/rfc2616/rfc2616.html.
To decide how to mix the operations of a workload, the user specifies the `mix` (lines #5-#20 of the mix specification). The mix allows the definition of sequences of operations, or Markov chains^ controlling how to navigate from one operation to another one, as well as the time the simulated user spends to think about the next operation to execute. The framework executing the test attempts to respect as much as possible the specified mix of operations and think time. Due to the indeterminism of software system behavior, a slight deviation from the same must be expected according to the known variance of SUT's behavior, the test execution environment, as well as the SUT deployment environments.
Full-size image for workloads: https://docs.google.com/drawings/d/1mQ5tfWB8APoxp_2T0gnwVvnWpBJtJxDjJr9eE_sfEwA
Full-size image for operation: https://docs.google.com/drawings/d/

11DmtReRXdtB_qtoZNT8MKRTUPxE_5u6gXmsDKLla1dg
Full-size image for mix: https://docs.google.com/drawings/d/10SATky5NdwlzMsjC3pcLbFAokzvkfLwNSNhMIXEkjHA

* For more details about the timing among operations refer to section 7 on the following guide http://faban.org/1.3/docs/guide/driver/elements.html
^ For more information about the possible mixes, refer to section 6 of http://faban.org/1.3/docs/guide/driver/elements.html. For more details about Markov chain specification, please refer to https://www.se.informatik.uni-kiel.de/en/research/projects/markov4jmeter

The System Under Test Specification

```
sut:
 name: { String : <name of the sut> }
 versions:
   # EITHER specific values
   values: { [String] : <name of versions in the wanted order> }
   # OR range
   range: { [String, String] : <a list of versions (inclusive)> }
 # OPTIONAL
 type: { String : <"wfms" | "http"> }
 sut_configuration:
   default_target_service:
     name: { String : <name of target service> }
     endpoint: { String : <base endpoint for each operation> }
     # OPTIONAL
     sut_ready_log_check: { String : <a regular expression to check the
     ↪  availability of the sut> }
   # OPTIONAL
   deployment:
     { String : <name of service> }: { String : <name of server alias> }
     ...
 # OPTIONAL
 services_configuration:
   { String: <name of service> }:
     # OPTIONAL
     resources:
       # OPTIONAL
       cpu: { [String] : <a list of values + unit> }
       # OPTIONAL
```

```
28        memory: { [String] : <a list of bytes value + unit> }
29      # OPTIONAL
30      configuration:
31        { String: <name of environment variable> }: { String: <value> }
32        ...
33    ...
```

Listing C.10. Expert Review: The SUT YAML Format Specification

In Listing C.10, we report the specification of the system under test and its configuration (`sut`). The users can specify the `name` (line #2) of the SUT and the target `versions` (lines #3-#7), as well as the `type` (line #9).
The SUT has to be configured to automate the test (lines #10-#19). The user configures the default target service for the workload and optionally states how to identify when the SUT is ready to receive the load (line #15). The proposed framework can also take care of deploying the SUT for different experiments, thus the user can specify data about the deployment of different services (lines #17-#19). If no deployment information is provided, the proposed framework selects the best suitable and available machines where to deploy the SUT, unless a full `endpoint` is specified. When a full endpoint is specified, the SUT is expected to be already deployed at the specified endpoint, thus the deployment and the service configurations (lines #21-#33) are skipped*.
The user can also specify service configurations (lines #21-#33) in terms of service' resources and configuration variables.
Full-size image: https://docs.google.com/drawings/d/1DwL95XXfXilsSsCX68hS71q_VtCNrPxwHSSYAojCCIk

* Please note that in order for complex tests to be executed, the SUT deployment is expected to be carried out by the proposed framework, so that configurations requested for different experiments can be injected at deployment time.

The Data Collection Specification

```
1  # OPTIONAL
2  data_collection:
3   only_declared: { Boolean : <"true" | "false"> } # default "false"
```

```
4   # OPTIONAL
5   services:
6     # IF collector does NOT require CONFIGURATION
7     { String : <name of service> }: { String | [String] : <names of BenchFlow
      ↪  collectors> }
8     # IF some collectors requires CONFIGURATION
9     { String : <name of service> }:
10      { String : <name of BenchFlow collector> }:
11        configuration:
12          { String : <name of environment variable> }: { [String] : <a list of
            ↪  possible values> }
13          ...
14      ...
15    ...
16  # OPTIONAL
17  workloads:
18    { String : <name of the workload> }:
19      # OPTIONAL IF trial execution framework = JMeter
20      jmeter:
21        interval: { String : <time interval for data collection><"s" (seconds)> } #
          ↪  default 1s
22      # OPTIONAL IF trial execution framework = Faban
23      faban:
24        interval: { String : <time interval for data collection><"s" (seconds)> } #
          ↪  default 1s
25    ...
```

Listing C.11. Expert Review: The Data Collection YAML Format Specification

In Listing C.11, we report the specification of the data collection services to be used to collect performance data used to compute the observed performance metrics.
Data collection services are available as part of the proposed solution and can collect client-side performance data (lines #17-#25), and server-side (lines #5-#15) resource utilization data, files produced by the SUT as well as dumps of databases (e.g., useful for computing BPMN20 WfMSs-specific metrics). The services can be configured (lines #9-#14) according to SUT configurations (e.g., to properly connect to a DBMS) and the list of services can be extended according to new requirements. Values of configuration variables can refer to keywords representing variables injected by the framework part of the proposed approach at runtime, to refer to, e.g., IP addresses and Ports of services from which data have to be collected, or configurations already

specified as part of the deployment descriptor part of the test bundle. Client-side collection services depend on the actual execution framework used to execute the experiments, and the user can specify the collection interval between two subsequent samplings of performance data. We currently support Faban (http://faban.org/) and JMeter (https://jmeter.apache.org/) as execution frameworks.
The proposed framework takes care of injecting collection services for the observed metrics, in case the user does not specify some of them and no custom configuration is required.
Full-size image: https://docs.google.com/drawings/d/1bJShriRttibBW8SmAStjxsBEFpOCHAYH4LIAniET3wE

Integrating Tests with a CSDL
The Test Suite Specification

```
version: { String : <Test Suite CDSL version, e.g. '1'> }
name: { String : <name of the performance test suite> }
# OPTIONAL
description: { String : <description of the performance test suite> }
suite:
  triggers:
    # OPTIONAL
    scheduled: { Boolean : <"true" | "false"> } # default "false"
    # OPTIONAL
    on:
      # OPTIONAL
      push:
        # OPTIONAL
        branches:
          - { String : <expression matching the name of branch> }
          ...
      # OPTIONAL
      pull_request:
        # OPTIONAL
        contexts: { String : <"head" | "merge" | "all"> }
        # OPTIONAL
        source_branches:
          - { String : <expression matching the name of branch> }
          ...
        # OPTIONAL
        target_branches:
          - { String : <expression matching the name of branch> }
          ...
```

```
29        # OPTIONAL
30        releases:
31          types: { [String] : <types of releases> }
32        # OPTIONAL
33        deployments: { [String] : <names of deployments> }
34    # OPTIONAL IF DEPLOYMENTS
35    environments:
36      - { String : <name of environments> }:
37          # OPTIONAL
38          skip_deploy: { Boolean : <"true" | "false"> } # default "false"
39      ...
40    tests: -
41      # EITHER a list of file paths containing tests
42      - { String : <path of file containing test> }
43      ...
44      # OR
45      include_labels: { [String] : <expression matching the name of labels to select
          ↪ tests> }
46
47  quality_gates:
48    criterion: { String : <"all_success" | "at_least_one_success"> }
49    # OPTIONAL
50    exclude:
51      - { String : <path of file containing test> }
52      ...
```

Listing C.12. Expert Review: The Tests Suite YAML Format Specification

In Listing C.12, we report the specification of the `test suite`. The user can specify test suites, to automatically select tests implemented using the specification presented above, in different moments of the CSDL lifecycle according to the generated events.
In CSDL software is usually committed to a versioning system by different users, and developed using multiple so-called feature branches. When features are considered mature enough, they are usually merged in a development version first and then in a production version of the SUT. These activities generate events such as `push`, `pull requests`, `releases` and `deployments`* (lines #10-#33), and these events are used to determine when to trigger different performance tests. In case of deployments, the user can also specify the environment of interest of such deployment to trigger the execution of performance tests (lines #35-#39) and indicate whether to skip the deployment of the SUT^ or not.

The selection of the tests realizing the test suite happens either using regular expressions matching `labels`, or by enumerating all the test specification files to execute (lines #40-#45). As per the test specification, also for test suites, the user is expected to define quality gates to determine the overall result of the test suite execution (lines #47-#52). The test suite can fail if at least one of the quality gates of the tests realizing the test suite fails (apart from the excluded ones as reported in lines #50-#52), or if none of the tests of the suite is successful. The second option is useful for example when testing different variants of the same SUT and it is sufficient that at least one is successful to proceed.
With the command line interface part of the proposed framework, the user can also submit information about the events generated as part of the CSDL, other than the tests and the test suites.
In case the test suite is not executed based on some events but scheduled to be executed, e.g., overnight, the user can specify so as well (line #8). The actual scheduling is expected to happen in the tools used for CSDL.
Full-size image: https://docs.google.com/drawings/d/1bW0cxksIjU6o-EEx7iZo0I-_t5mpj3bt2rGrvlfQO8E

* For more information about CSDL and the mentioned events refer to https://nvie.com/posts/a-successful-git-branching-model/
^ E.g., if it is already deployed by the CSDL, in such case the endpoint of the executed tests are expected to be specified and executed tests are not expected to requiring SUT configuration other than the default one
NOTE: termination criteria are not required for the integration with CSDL, because each single test part of the test suite is expected to define proper termination criteria.

C.4.2 Examples of Declarative Performance Tests Specification and Integration with CSDL

Example of a Load Test

```
1 version: "3"
2 name: "Load Test"
3 description: "Example of Load Test"
4 labels: "load_test"
5 configuration:
```

```yaml
  goal:
    type: "load_test"
    observe:
      workloads:
        workload_a: avg_response_time, avg_latency
        workload_b: avg_response_time, avg_latency
        workload_b.operation_a: avg_response_time
      services:
        service_a: avg_cpu, avg_memory
        service_b: avg_cpu, avg_memory
        dbms_a: avg_cpu, avg_memory, avg_io
  load_function:
    users: 1000
    ramp_up: 5m
    steady_state: 20m
    ramp_down: 5m
  termination_criteria:
    test:
      max_time: 40m
  quality_gates:
    workloads:
      workload_a:
        - max_mix_deviation: 5%
          max_think_time_deviation: 2%
          gate_metric: avg_response_time
          condition: "<="
          gate_threshold_target: "100ms"
          gate_threshold_minimum: "200ms"
sut:
  name: "my_app"
  versions:
    values: "v1.5"
  type: "http"
  sut_configuration:
    default_target_service:
      name: "service_a"
      endpoint: "/"
      sut_ready_log_check: "/(.*)System started(.*)/g"
    deployment:
      service_a: "my_server"
  services_configuration:
    service_b:
      resources:
        cpu: 200m
        memory: 256Mi
      configuration:
```

```
52        THREADPOOL_SIZE: 64
53  workloads:
54    workload_a:
55      item_a:
56        driver_type: "http"
57        inter_operation_timings: "negative_exponential"
58        popularity: 80%
59        operations:
60          operation_a:
61            protocol: "https"
62            endpoint: "/"
63            method: "GET"
64            extract_regex:
65              - title:
66                pattern: "<title>(.*?)</title>"
67                default: ""
68                match_number: 1
69          operation_b:
70            protocol: "https"
71            endpoint: "/${title}"
72            method: "POST"
73            body_file: "datasource_a"
74          data_sources:
75            - path: "/path_to_datasource_a"
76              delimiter: ","
77              name: datasource_a
78              retrieval: "random"
79          mix:
80            max_deviation: 5%
81            flat: "75.0% tt(1000.0 500.0), 25.0% tt(2000.0 400.0)"
82      item_b:
83        driver_type: "http"
84        inter_operation_timings: "negative_exponential"
85        popularity: 20%
86        operations:
87          operation_a:
88            protocol: "https"
89            endpoint: "/"
90            method: "GET"
91    workload_b:
92      item_a:
93        driver_type: "http"
94        target_service: "service_b"
95        operations:
96          operation_a:
97            protocol: "https"
```

```
98            endpoint: "/"
99            method: "GET"
100 data_collection:
101   # AUTOMATICALLY attached based on the observe section IF NOT specified
102   services:
103     service_a: "stats"
104     service_b: "stats"
105     dbms_a: "stats"
```

Listing C.13. Expert Review: A Load Test for a Web service YAML Example

In Listing C.13, we report a complete example of a load test. The load test in the example states:

a) the load test goal, being the specified test a load test (line #7)

b) the metrics to observe, on workloads, operations, and services (lines #8-#16)

c) the target SUT and its configuration (lines #34-#52)

d) the load function (lines #17-#21)

e) the termination criterion for the test, based on maximum execution time (lines #22-#24)

f) the quality gates, referring to the workloads named "workload_a" and based on client-side performance metrics (lines #25-#33)

g) the two workloads, operations and mixes issued to the SUT and their popularity (lines #53-#99)

h) the data collection services, that in this case can also be automatically attached by the proposed framework (lines #100-#105)

The execution of such a load test is expected to last at most 40 minutes and compute the specified observed metrics.
Full-size image: https://docs.google.com/drawings/d/1vF8FaSNO4BR25yXSVrssVKo80lgTxbGPPvHn0KSVVsY

Example of a Smoke Test

```yaml
version: "3"
name: "Smoke Test"
description: "Example of Smoke Test"
labels: "smoke_test"
configuration:
  goal:
    type: "smoke_test"
    observe:
      ...
  load_function:
    users: 10
    ramp_up: 30s
    steady_state: 2m
    ramp_down: 30s
  termination_criteria:
    test:
      max_time: 20m
    ...
  quality_gates:
    ...
sut:
  ...
workloads:
  ...
data_collection:
  # AUTOMATICALLY attached based on the observe section IF NOT specified
  services:
    ...
```

Listing C.14. Expert Review: A Smoke Test for a Web service YAML Example

In Listing C.14, we report an example of a smoke test. We omit many details in the specification, assuming they are the same as the previous load test. This is something we allow as part of the proposed solutions, to facilitate reusability of basic tests and opinionated templates defined by expert users and/or provided as part of our approach. A user could define basic tests, including all the details about the workload, the SUT, collections services, etc and then override such specifications with an additional specification providing only the delta to be applied in addition (if not present in the basic specification) or in substitution (if present in the basic specification) information*.
In the provided example for the smoke test, assuming the user already defined the previous load test, she has only to specify:

a) the updated goal type to “smoke_test” (line #7)

b) the updated load function (lines #10-14) and termination criterion (lines #15-#17) because smoke tests usually need less time than load tests to be executed

The execution of such a smoke test is expected to last at most 20 minutes.
Full-size image: https://docs.google.com/drawings/d/1xgNGPG_6fgKKgDim1F2yxnC2M9FlyB7YUAkrxhKSSM0

* When submitting the test using the CLI provided as part of the approach, the user can specify multiple tests as part of the test bundle and they are merged overriding information from the first to the last in order of specification.

Example of a Configuration Test

```yaml
version: "3"
name: "Configuration Test"
description: "Example of Configuration Test"
labels: "configuration"
configuration:
  goal:
    type: "configuration"
    stored_knowledge: "true"
    observe:
      ...
    exploration:
      exploration_space:
        services:
          service_a:
            resources:
              cpu:
                range: [100m, 1000m]
                step: "*2"
              memory:
                range: [256Mi, 1024Mi]
                step: "+256Mi"
            configuration:
              NUM_SERVICE_THREAD: [12, 24]
          dbms_a:
            resources:
              cpu:
                range: [100m, 1000m]
```

```
28                step: "*2"
29              memory:
30                range: [256Mi, 1024Mi]
31                step: "+256Mi"
32            configuration:
33              QUERY_CACHE_SIZE: [48Mi, 64Mi, 88Mi, 112Mi]
34      exploration_strategy:
35        selection: "one_at_a_time"
36  load_function:
37    ...
38  termination_criteria:
39    ...
40  quality_gates:
41    ...
42 sut:
43   ...
44 workloads:
45   ...
46 data_collection:
47   ...
```

Listing C.15. Expert Review: A Configuration Test for a Web service YAML Example

In Listing C.15, we report an example of a configuration test. As for the smoke test, we omit many details assuming they are the same as the load test, except for the termination criteria that are automatically determined by the proposed framework considering the number of tests in the exploration space and the load function. The configuration test in the example states:

a) the goal as being a configuration test (line #7)

b) re-utilization of stored knowledge (line #8), important to reduce the execution time

c) the exploration space, being this a configuration test thus requiring the specification of the space of configuration to explore (lines #11-#33). The exploration space covers multiple services resources and configurations

d) the exploration strategy (lines #34-#35) as one selecting one experiment at a time in the exploration space following the order of dimensions and value specified in the exploration space section

The execution of such a configuration test is expected to last until the entire space is explored.

Full-size image: https://docs.google.com/drawings/d/1gsutaAWDgZG_99akSmJUigS0ySeMGwVlKI_5sg0IaQM

Example of a Regression Test

```yaml
version: "3"
name: "Regression Test"
description: "Example of Regression Test"
labels: "regression_complete"
configuration:
  goal:
    type: "regression_complete" # OR regression_intersection
    stored_knowledge: "true"
    observe:
      workloads:
        workload_a: avg_response_time
        workload_b: avg_response_time
  load_function:
    users: 1000
    ramp_up: 5m
    steady_state: 20m
    ramp_down: 5m
  termination_criteria:
    test:
      max_time: 3h
    experiment:
      max_failed_trials: 0%
      workloads:
        workload_a:
          confidence_interval_metric: avg_response_time
          confidence_interval_value: 200ms
          confidence_interval_precision: 95%
  quality_gates:
    regression:
      workload: workload_a
      gate_metric: avg_response_time
      regression_delta_absolute: 50ms
    workloads:
      workload_a:
        - max_mix_deviation: 5%
          max_think_time_deviation: 2%
```

```yaml
          gate_metric: avg_response_time
          condition: "<="
          gate_threshold_target: "150ms"
          gate_threshold_minimum: "250ms"
sut:
  name: "my_app"
  versions:
    values: [ "v1.5", "v1.6" ]
  type: "http"
  sut_configuration:
    ...
  services_configuration:
    service_a:
      resources:
        cpu: 500m
        memory: 512Mi
      configuration:
        THREADPOOL_SIZE: 64
    service_b:
      resources:
        cpu: 200m
        memory: 256Mi
      configuration:
        THREADPOOL_SIZE: 64
    dbms_a:
      resources:
        cpu: 800m
        memory: 512Mi
      configuration:
        QUERY_CACHE_SIZE: 88Mi
workloads:
  ...
data_collection:
  ...
```

Listing C.16. Expert Review: A Regression Test for a Web service YAML Example

In Listing C.16, we report an example of a regression test. The omitted sections are expected to be the same as the previous load test. The regression test in the example states:

a) the goal as being a regression test considering the entire workloads' operations (line #4)

b) re-utilization of stored knowledge (line #8), important to reduce the execution time and retrieve previous executions of the same regression test

c) the metrics of interest on which the regression is going to be evaluated (lines #9-#12)

d) the load function (lines #13-#17). The exploration space covers multiple services resources and configurations

e) the termination criteria for the test (lines #18-#27). In the case of the example, both test and experiment termination criteria are specified. For what concern experiment termination criteria is specified that no experiments can fail and that workload names "workload_a" has to have the average response time metric within the provided confidence interval. This request is consistent with the following stated regression criteria

f) the quality gates (lines #28-#40), consisting of 1) the regression criteria (lines #29-#32) defining the maximum expected regression for the response time of the "workload_a" in absolute term, and 2) the quality gate on the same workload controlling the range of the same metric as the regression criteria to ensure it is below a defined threshold

g) the SUT versions (lines #43-#44) and the services configuration (lines #48-#66) to ensure a stable configuration for all the services part of the SUT for all the regression tests

The execution of such a regression test is expected to last at most 3 hours and provide results about a regression being present between version "v1.5" and version "v1.6" of the target SUT.
Full-size image: https://docs.google.com/drawings/d/1wUBK9ibk1wDzGCw0j1fqrPrI0Ga1MSbWlh1cxabCztc

Example of Integration with CSDL

```
1 version: "1.6"
2 name: "Smoke Tests on Features"
3 description: "Runs all the Smoke Tests on new features merging to the Development
  ↪ branch"
4 suite:
```

```
5   triggers:
6     on:
7       pull_request:
8         contexts: "merge"
9         source_branches:
10          - "feat-*"
11        target_branches:
12          - "development"
13  tests:
14    include_labels: [ "(.*)smoke_test(.*)" ]
15
16 quality_gates:
17   criterion: "all_success"
18   exclude:
19     - "./tests/performance/smoke/smoke_test_alternative_scenario.yaml"
```

Listing C.17. Expert Review: A Test Suite YAML Example

In Listing C.17, we report an example of test suite specification to integrate the execution of tests in CSDL. The proposed example refers to smoke tests the user wants to execute on developed feature branches. The test suite in the example states:

a) the triggers (lines #5-#12) defining the execution of the suite based on pull requests from feature branches to the development branch. The context indicated is "merge" meaning the test is requested to be executed on the version of the SUT built out of the feature branch codebase merged with the development codebase

b) the selection criteria of the tests to be included as part of the test suite (lines #13-#14). Selection criteria are based on the labels, selecting all the tests having as part of the labels the string "smoke_test" in order of file name under a "./test/performance" folder part of the same code repository where the SUT is developed. If no tests with the specified labels are found the test suite can not be executed

c) the quality gates (lines #16-#19) of the test suite. In the case of the example, the quality gates expect all the executed test to be successful, but the one reported in the `exclude` block (lines #18-#19)

The execution of such a test suite is expected to last until all the selected tests are executed.

Full-size image: https://docs.google.com/drawings/d/1eOlDELcfr5aEEX35KSO8JzYLOXWjoBKOs2Ug5Sl4B84

C.5 Review of the Approach

We are asking you to state your opinion about the overall approach. For your convenience, we provide a PDF version of the "Overview of the Approach and Examples": https://drive.google.com/file/d/1e-Vso1k9SgiAsfBnosec70rcShvvTbdm

C.5.1 Expressiveness

7a. How do you evaluate the overall expressiveness of the proposed DSL?
When referring to the context of performance testing definition and automation, particularly in CSDL, report how you evaluate the overall expressiveness of the DSL proposed in this survey.
Very Poor ☐—☐—☐—☐—☐ Excellent

7b. How do you motivate the previous answer about expressiveness?:

_____

C.5.2 Usability and Effort

How do you compare the perceived usability and effort of the proposed approach for performance test automation compared to the perceived usability and effort of standard imperative approaches?
When referring to the usability and the effort of standard imperative approaches, such for example the one proposed by JMeter, report how you compare the perceived usability and effort to define performance tests and control their automation processes of the approach proposed in this survey concerning the perceived usability and effort of standard imperative approaches

8a. Perceived Usability: Proposed Approach VS Standard Imperative
Much worse ☐—☐—☐—☐—☐ Much better

8b. Perceived Effort: Proposed Approach VS Standard Imperative
Much worse ☐—☐—☐—☐—☐ Much better

8c. How do you motivate the previous answer about usability and effort?:

C.5.3 Reusability

9a. How do you compare the perceived overall reusability of performance tests implemented using the proposed approach compared to the reusability of tests implemented using the standard imperative approaches?
When referring to the reusability of standard imperative approaches, such for example the one proposed by JMeter, report how you compare the perceived reusability of performance tests implemented using the approach proposed in this survey compared to the perceived reusability of tests implemented using standard imperative approaches
Much worse ☐—☐—☐—☐—☐ Much better

9b. How do you motivate the previous answer about reusability?:

C.5.4 Suitability for the Target Users

10a. How suitable do you consider the proposed approach for the target users of the same?
When referring to the target users of our approach, namely developers, software testers, quality assurance engineers, and operations engineers, report how suitable you consider the proposed approach for them.
Not suitable ☐—☐—☐—☐—☐ Very suitable

10b. How do you compare the suitability of the proposed approach compared to the suitability of standard imperative ones for the target users of the proposed approach?
When referring to the suitability of standard imperative approaches, such for example the one proposed by JMeter, for the target users of the proposed approach, report how you compare the suitability of the approach proposed in this survey to the suitability of standard imperative approaches for the same users.
Much worse ☐—☐—☐—☐—☐ Much better

10c. How do you motivate the previous answers about the suitability of the approach for the target users?:

C.6 Final Evaluations

11a. What are the main PROs of the proposed approach?:

11b. What are the main CONs of the proposed approach?:

11c. Name similar approaches you are aware of:

C.7 Thank you!

As anticipated at the beginning of the survey, we value the time you spent on providing the feedback, so we want to give you back something to show our appreciation.

We are giving away three little gifts as a sign of appreciation, raffled among all the participants completing the survey. The three gifts are all the same, the value is € 20 each, and at most one can be assigned to each participant. They are going to be discount coupons for well-known websites providing services and products.

The raffle is going to happen by the middle of August 2020, and only if at least 15 complete surveys are submitted*. To increase the probability of the raffle to happens, please share the survey with people you know that might contribute with valuable answers according to the defined target audience for this survey. Sharing the survey is very helpful for us.

The raffle is going to be executed on `https://commentpicker.com/random-name-picker.php` (with anonymized emails) so that we can provide you back with proof of results.

Leave your email address if you want to participate in the raffle. We are going to write you back when the raffle results are available, and if you are one of the lucky winners, we are going to attach to the email the gift as well.

* You were able to check the current number of submitted responses by clicking on “See previous responses” on the next screen after clicking on the “Submit” button. I had to disable this because responses might contain sensitive data, such as the email address. Status at 31.07.2020: 18 responses.

12. What is your email address?:

_____

Appendix D

Summative Evaluation Survey

D.1 Declarative Performance Testing

The Proposed Approach for Declarative Performance Tests Specification and Execution
This approach is part of a PhD contribution. We ask you to contribute with your valuable feedback to complete the evaluation section of the same. The answers are going to be reported anonymously as part of the thesis. We value the time you spend on providing the feedback, so we want to give you back something to show our appreciation. We are giving away three little gifts as a sign of appreciation, raffled among all the participants completing the survey. More details on this at the end of the survey, where you can decide whether to opt-in or not.

NOTE 1: due to limitations of Google Forms, we can not use a larger font size. To increase the font-size please follow the advice on the following guide: `https://www.computerhope.com/issues/ch000779.htm` (Suggested: use the topmost Tip of the referenced guide). Images are not going to be increased guaranteeing the quality, thus we always offer a full-size image link.
NOTE 2: going back to the previous page during the survey, usually saves the already provided answers when coming back to the page containing the answer. If you think you might need to do this, we advise you to test the behavior with your browser to be sure about it, before completing multiple answers

D.2 Summative Evaluation of The Proposed Approach

Context
The primary objective of performance testing is to assess the performance of a system, to ensure it is compliant with specified service level requirements. To accomplish so, multiple kinds of tests have to be executed to ensure all the requirements are met. Recent trends in the industry show increasing adoption of Development and Operations (DevOps) practices. Alongside the adoption of DevOps, performance testing continues to evolve to meet the growing demands of the modern enterprise and its need for automation. Modeling and automated execution of performance tests are time-consuming and difficult activities, requiring expert knowledge, complex infrastructure, and a rigorous process to guarantee the quality of collected performance data and the obtained results. Currently available performance testing approaches are not well integrated with DevOps practices and tools and are often focusing only on specific needs of performance testing modeling and automation.

Proposal
We propose a **Declarative Approach for Performance Tests Execution**, enabling the continuous and automated execution of performance tests alongside the Continuous Software Development Lifecycle (CSDL)*.
The approach is comprised of a declarative Domain Specific Language (DSL) enabling the declarative specification of performance tests and their automated orchestration processes alongside the CSDL, and a framework for end-to-end automated performance testing relying on the contributed DSL. The main target systems are RESTful Web services and Business Process Model and Notation 2.0 (BPMN20) Workflow Management Systems (WfMSs) deployed using a deployment descriptor relying on Docker containers (More details on https://docs.docker.com/compose/compose-file/).
The main target users of our approach are developers, software testers, quality assurance engineers, and operations engineers. The users of our approach can specify performance tests using a goal-oriented approach, and declaratively configure the entire process followed to reach the goal and evaluate the test result using the same DSL. With the same DSL, they can build test suites and define when different tests are executed during the CSDL, according to specific triggers. The DSL is used to configure a framework programmed using such DSL and implementing the processes needed for executing performance

experiments part of the declared test, and a control loop watching the state of the executing test and scheduling experiments to reach the specification stated in the declarative specification submitted by the users.
More details are provided in a dedicated section of this survey.

Summative Evaluation Objective
The objective of this summative evaluation is to collect feedback about the proposed approach in terms of **expressiveness** and **reusability** of the DSL, as well as **learnability** and overall perceived **usability** of the same, especially comparing it to standard imperative approaches+, like the one proposed by JMeter, and considering the target users of the approach^.

Summative Evaluation Target Audience
We mainly target developers, software testers, quality assurance engineers, and operations engineers, both in academia and industry. If you are not part of the target audience, but you are an IT practitioner, go ahead with the survey anyway. Your feedback is valuable as well.

Structure of the Survey
The survey starts with a section where we ask you to fill in some information about your experience and expertise. We then present you the approach in a dedicated section where we go over all the elements of the proposed DSL, and we provide you with examples illustrating how a user is expected to state goal-driven declarative performance tests using our approach. After an introduction to the approach, we are going to ask you some multiple-choice questions about the same where you can also optionally provide additional feedback. Finally, we propose some questions to collect your general opinion, PROs, and CONs of the approach, and other concluding questions.

Expected average completion time: up to 35-50 minutes, of which up to 25-30 minutes for learning about the proposed approach, depending on your background.

The survey is anonymous and no contact details are collected, unless provided by you at the end of the survey. Constructive feedback is always the most valuable.
* The approach has been first presented at the International Workshop on Quality-Aware DevOps in 2017 (paper: https://dl.acm.org/doi/pdf/

10.1145/3053600.3053636, presentation: https://bit.ly/2Vtcqny) and then at the International Conference on Performance Engineering in 2018 (paper: https://dl.acm.org/doi/pdf/10.1145/3184407.3184417, presentation: https://bit.ly/3g7T7YP). It has been largely extended over time.
+ A brief overview of imperative approaches vs declarative approaches for testing can be found on https://wiki.saucelabs.com/display/DOCS/Best+Practice%3A+Imperative+v.+Declarative+Testing+Scenarios
^ Evaluating the framework allowing the automated execution of the declaratively specified performance tests is out of the scope for this survey since it is a research prototype and it is not widely used as other frameworks for performance testing.

D.3 Experience and Expertise

In this section, we are collecting some information about you, to better analyze the answers you are going to provide in the following sections.

D.3.1 Education

1. **Latest degree of education**
   - ☐ Middle school.
   - ☐ High School.
   - ☐ Bachelor Degree.
   - ☐ Master Degree.
   - ☐ PhD.
   - ☐ Other.

D.3.2 Professional Role

2. **Your current professional role**
   - ☐ Software Developer.
   - ☐ Software Engineer.
   - ☐ Software Tester.
   - ☐ DevOps Engineer.
   - ☐ QA Analyst.

- ☐ Performance Analyst.
- ☐ Site Reliability Engineer.
- ☐ Researcher.
- ☐ Professor.
- ☐ Student.
- ☐ Other: _____.

D.3.3 Experience in the current professional role

3. Years of experience in the current professional role: _____

D.3.4 Experience related to software performance testing

4. Years of experience in the area of software performance testing. Not necessarily as the main activity. NOTE: it can be 0. : _____

D.3.5 Which are your main tasks and activities?

5. Describe your main tasks and activities in your current role:

_____

D.3.6 How familiar are you with the following concepts?

State how familiar are you with the following concepts related to performance test implementation and execution in the context of this survey

6a. Performance Testing Not at all ☐—☐—☐—☐—☐ Expert

6b. System Under Test Not at all ☐—☐—☐—☐—☐ Expert

6c. System Workload Not at all ☐—☐—☐—☐—☐ Expert

6d. Performance Data Collection Not at all ☐—☐—☐—☐—☐ Expert

6e. System Performance Metrics Not at all ☐—☐—☐—☐—☐ Expert

6f. Performance Tests Execution Framework Not at all ☐—☐—☐—☐—☐ Expert

6g. Load Driver Not at all ☐—☐—☐—☐—☐ Expert

6h. Imperative Approaches for Performance Engineering (e.g., JMeter) Not at all ☐—☐—☐—☐—☐ Expert

6i. Declarative Performance Engineering
Not at all □—□—□—□—□ Expert

6j. System Deployment Descriptor Not at all □—□—□—□—□ Expert

6k. Docker Container Not at all □—□—□—□—□ Expert

6l. Container Orchestrator Framework (e.g., Kubernetes)
Not at all □—□—□—□—□ Expert

6m. Continuous Integration and Delivery
Not at all □—□—□—□—□ Expert

6n. RESTful Web Services Not at all □—□—□—□—□ Expert

6o. YAML Language Not at all □—□—□—□—□ Expert

D.4 Overview of the Approach and Examples

In the following, we present an overview of the proposed approach. We illustrate the YAML specification format the users of our approach are expected to master for declaratively specifying performance tests and configuring the test's automation process. We then present some examples of declarative performance test specifications relying on the proposed YAML specification format. What users are expected to state for declarative performance test automation To rely on the proposed declarative performance test automation solution, users need to have a basic understanding of the performance testing domain* and are expected to define a test using the proposed YAML specification format and specify:

1) The test goal and its configuration, as well as the performance space to be explored if required by the goal (`goal`, `exploration_space`)

2) The load function (`load_function`)

3) The performance metrics of interest (`observe`)

4) The criteria to determine if the test execution has to be terminated before its completion (`termination_criteria`)

5) The expected outcome of the test according to defined quality criteria (`quality_gates`)

6) The workloads used for the test and how to mix its operations (`workloads`, `operations`, `mix`)

7) The system under test type, and its configuration (`sut`)

8) The data collection services needed to collect performance data useful to compute the metrics of interest (`data_collection`)

After specifying different tests, users can also define test suites to integrate subsets of tests as part of the CSDL. To do so, users are expected to define:

1) The test suite, selecting tests by referring to YAML files defining them or by selecting them using labels (`suite`, `tests`)

2) The triggers activating the test execution, according to CSDL events (`triggers`)

3) Optionally, the environments on which tests have to be executed (`environments`)

4) The expected outcome of the test suite according to defined quality criteria (`quality_gates`)

Users are expected to provide the test specifications, as well as test data and a system under test (SUT) deployment descriptor as part of a so-called *test bundle*. The SUT deployment descriptor defines the services part of the SUT, their configuration, deployment specification, and how they communicate among them. More details are provided on the official reference: https://docs.docker.com/compose/compose-file/.
The proposed framework provides a command-line interface (CLI) that can also be integrated into different Continuous Integration and Continuous Delivery pipelines, defined with tools such as Jenkins (https://www.jenkins.io/). Through such CLI the user can automatically submit tests and test suite specifications for execution to the proposed framework.
Test specifications can be reused, extended and overridden, to facilitate reuse of base test definitions for more complex tests. Some examples are provided in the examples section, presented after the YAML specification format.
In the following, we present the YAML specification format as well as examples of declarative performance tests specification and integration with CSDL.

* Note that this statement refers to performance test specification. For what concerns analyzing the performance results, and improving the system performance basic understanding of the performance domain might not be sufficient.

D.4.1 YAML Specification Format

Defining a Declarative Performance Test
Overall Test Specification

```
version: { String : <Test DSL version, e.g. '1'> }
name: { String : <name of the performance test> }
# OPTIONAL
description: { String : <description of the performance test> }
# OPTIONAL
labels: { [String] : <a list of labels> }
configuration:
  goal:
    ...
  load_function:
    # OPTIONAL IF SPECIFIED IN EXPLORATION
    users: { Number : <total number of users to be simulated> }
    ramp_up: { String : <amount of time><unit> }
    steady_state: { String : <amount of time><unit> }
    ramp_down: { String : <amount of time><unit> }
  # OPTIONAL
  termination_criteria:
    ...
  # OPTIONAL
  quality_gates:
    ...
sut:
  ...
workloads:
  ...
# OPTIONAL
data_collection:
  ...
```

Listing D.1. Summative Evaluation: The Test YAML Format Specification

In Listing D.1, we report the specification of the overall test structure. As you can see, the user can specify all the information described above. According to the selected goal, the user is requested to configure the automation process followed to reach such goals.
For what concerns the `load_function`, the user can specify the wanted number of simulated users and characteristics of the load function in terms of time to go from zero to the number of specified users (`ramp_up`), time to issue the load

to the system under test at the specified number of users (`steady_state`), and time to go from the wanted number of users back to zero (`ramp_down`).*
Among the other data, the user can also specify `labels` (line #6). Labels are important for selecting different test specifications when integrating test execution in CSDL.
Full-size image: https://docs.google.com/drawings/d/1a9CSMBnDm8ToNRbKpj-Vb7Q7oxVR71T0V4E3ZBC1ba4

* We currently only support the mentioned shape of the load function, due to limitations in the frameworks we use to automate the performance test execution. Other frameworks can be integrated, thus the shapes of supported load functions could be enhanced.

The Goal Specification

```
1  configuration:
2   goal:
3     type: { String : < "load" | "smoke" | "sanity" | "configuration" |
       ↪  "scalability" | "spike" | "exhaustive_exploration" | "stability_boundary" |
       ↪  "capacity_constraints" | "regression_complete" | "regression_intersection"
       ↪  | "acceptance" > }
4     # OPTIONAL
5     stored_knowledge: { Boolean : <"true" | "false"> } # default "false"
6     observe:
7       ...
8     # OPTIONAL
9     exploration:
10      ...
```

Listing D.2. Summative Evaluation: The Goal YAML Format Specification

In Listing D.2, we report the specification of the performance test goal. As you can see, the user can specify many different supported goals (line #3), such as `load test`, `smoke test`, `configuration test`, and `regression test`. The goal is usually mapped to well-known performance test types. The number of supported goals can be extended over time.
Among the other data, the user can also specify whether she wants to use `stored_knowledge` (line #4) or not. Reusing stored knowledge enables the possibility to reuse previous results of tests and experiments defining the exact

same specification, executed as part of previous test execution, or as part of other tests specifications scheduled in parallel to the submitted one.
Full-size image: https://docs.google.com/drawings/d/1aEFxdmRDOFVeRC0OZRznZj-1saAX_SoVvDKuCTFkV6Y

* The actual values for the type of test can vary. Usually, also the variant of the name with the addition of the suffix "_test" is supported.

The Exploration Space Specification

```
goal:
  # OPTIONAL
  exploration:
    exploration_space:
      services:
        { String : <name of service> }:
            # OPTIONAL
            resources:
              cpu:
                # EITHER specific values
                values: { [String] : <a list of values + unit> }
                # OR range
                range: { [String, String] : <a list of values + unit (inclusive)>
                ↪ }
                # IF range we can specify step
                step: { String : <step between the values in the range as a
                ↪ mathematical expression: +,-,*,/,^><values + unit> }
              memory:
                # EITHER specific values
                values: { [String] : <a list of bytes value + unit> }
                # OR range
                range: { [String, String] : <a list of bytes value + unit
                ↪ (inclusive)> }
                # IF range we can specify step
                step: { String : <step between the values in the range as a
                ↪ mathematical expression: +,-,*,/,^><bytes value + unit> }
            # OPTIONAL
            configuration:
              { String : <name of environment variable> }: { [String] : <a list of
              ↪ possible values> }
              ...
        ...
      load_function:
```

```
29        ...
30      # OPTIONAL IF goal.type = stability_boundary
31      stability_criteria:
32        services:
33          { String : <name of service> }:
34            # OPTIONAL
35            avg_cpu: { String : <a mathematical expression:
              ↪  >,<,>=,<=,=><number><"%"> }
36            # OPTIONAL
37            avg_memory: { String : <a mathematical expression:
              ↪  >,<,>=,<=,=><number><"%"> }
38          ...
39        workloads:
40          { String : <name of workload> }:
41          ...
42    exploration_strategy:
43      selection: { String : <"one_at_a_time" | "random_breakdown" |
        ↪  "stability_boundary_first"> }
44      # OPTIONAL
45      validation: { String : <random_validation_set> }
46      # OPTIONAL
47      regression: { String : <mars> }
```

Listing D.3. Summative Evaluation: The Exploration Space and Exploration Strategy YAML Format Specification

In Listing D.3, we report the specification of the exploration space required by some of the supported goals, such as the `configuration test`.
When a complex test is specified, the user can define the exploration space in terms of `services' resources`, and `configuration` (a map of key-value pairs), and the `load function`. Services refer to services defined in the system under test deployment descriptor, provided as part of the test bundle.
For some goals, specific data have to be specified, such as for the "`stability_boundary`" one. In such cases, specific sections of the exploration space are provided. For the stability boundary test, the `stability criteria`, section is provided (lines #36-#47), allowing the user to specify stability criteria on services and workloads.
The exploration space could vary in size, thus the user can rely on a specification using a `range`, (e.g., line #13) of values to explore and a `mathematical function`, (e.g, line #15) to navigate the specified range.
Our approach computes the entire performance space to explore calculating

the combinations without repetitions on all the possible values specified by the user in the configuration space, considering each specified dimension as a vector. For example, if the user defines an exploration space with a "service_a" having as resources value for memory: [128Mi, 256Mi] and a load function defining 100 as the number of users, then the exploration space is comprised of two experiments, one with 100 users and 128Mi of memory for "service_a", and a second one of with 100 users and 256Mi of memory for "service_a".
Given that the exploration space could explode in terms of experiments to be executed, we provide different strategies to explore the space (lines #48-#53). We also alert the user if the size of the space is greater than a threshold (default: 10)*. One of the possible strategies that can be specified, is a predictive strategy (lines #51-#53) attempting to approximate the performance of the SUT in the performance, space, thus reducing the number of executed experiments.
Full-size image: https://docs.google.com/drawings/d/1ah5nazm99zRtgJmb-JXNbyLkHc7VAHHwinJ9iSYrluc

* Users of the framework executing the specified test can specify the threashold as part of the framework configurations.

The Specification for the Performance Metrics to Observe

```
1  goal:
2    observe:
3     # OPTIONAL
4     workloads:
5       # OPTIONAL
6       { String : <name of workload> }: { [String] : <a list of workload metrics> }
7       # OPTIONAL
8       { String : <name of workload.operation> }: { [String] : <a list of workload
          ↪  metrics> }
9       ...
10    # OPTIONAL
11    services:
12      { String : <name of service> }: { [String] : <a list of service metrics> }
13      ...
```

Listing D.4. Summative Evaluation: The Observe YAML Format Specification

In Listing D.4, we report the specification of the performance metrics of interest for the performance test. The user can request to `observe` a wide range of performance metrics and statistics provided by the proposed framework and computed on collected performance data, by referring to them by their names. Performance metrics can be observed of both the `workloads`, representing client-side performance metrics, as well as for the `services`, representing server-side performance metrics. Examples of workload metrics are average response time and average latency. Examples of service metrics are average CPU utilization and maximum memory utilization. We also provide SUT-specific metrics, for example for BPMN20 WfMSs we provide custom metrics about number and time of executed process instances, as well as other ones (for more details, refer to section 4 of http://www.pautasso.info/biblio-pdf/benchflow-bpmds2017.pdf)
Full-size image: https://docs.google.com/drawings/d/1T2nd0I9EpDSoM3gVRV0n1BYDwHrHusNv8F8HOLaUZjI

The Termination Criteria Specification

```
configuration:
 # OPTIONAL
 termination_criteria:
   # OPTIONAL
   test:
     max_time: { String : <amount of time><unit> }
     # OPTIONAL
     max_number_of_experiments: { Number : <maximum number of experiments> }
     # OPTIONAL
     max_failed_experiments: { String : <number><"%"> }
   # OPTIONAL
   experiment:
     max_number_of_trials: { Number : <maximum number of trials> }
     # OPTIONAL
     max_failed_trials: { String : <number><"%"> }
     # OPTIONAL
     workloads:
       # OPTIONAL
       { String : <name of workload> }:
         confidence_interval_metric: { String : <a workload metrics> }
         confidence_interval_value: { Number : <value of the workload metric> }
         confidence_interval_precision: { String : <number><"%"> }
       ...
     # OPTIONAL
```

```
25      services:
26        { String : <name of service> }:
27          confidence_interval_metric: { String : <a service metrics> }
28          confidence_interval_value: { Number : <value of the service metric> }
29          confidence_interval_precision: { String : <number><"%"> }
30        ...
```

Listing D.5. Summative Evaluation: The Terminationa Criteria YAML Format Specification

In Listing D.5, we report the specification of the criteria used to determine if the test execution has to be terminated before its completion. Given performance tests may need a considerable amount of time to be executed, the support of tests exploring potentially vast performance space, and the intrinsic fact performance test can fail or have non-deterministic issues during execution (e.g., due to instability of the SUT, the load drivers, or the performance test execution framework), it is of utmost importance to specify criteria controlling when to stop tests that are expected not to be successful.
A termination criterion can be specified on the `test` itself (lines #5-#10), or on `experiments` (lines #12-#30). At the test level, the users can state the maximum time the test is allowed to be executed*, as well as upper bounds for the number of executed experiments or the number of failed experiments. At the experiment level, the user can specify the maximum number of trials (e.g., the number of repetitions a single experiment is allowed to perform to provide reliable performance metrics) as well as the maximum number of failing trials. She can then define termination criteria based on confidence intervals of workloads and services metrics. The confidence interval is then used to dynamically compute the number of trials+.
Termination criteria are evaluated during a test and experiment execution. The result is determined by evaluating every single criterion for the given entity and then applying the OR operator to the criterion conditions result. As soon as one criterion results verified for the given entity, the execution is terminated. For example, for a time-based test termination criterion, as soon as the criterion is met the test is terminated. For termination criterion based on confidence intervals for experiments, as soon as the criterion results valid, the iteration of the experiment in multiple trials is terminated.
Full-size image: https://docs.google.com/drawings/d/16uJHecrkh9ML_vUrFlR1b_YwJ-1V43yE1pJNss-Ji7w

* the proposed frameworks checks for consistency of the termination criteria, with the number and time needed for executing the experiments part of the test and warns the user in case of inconsistencies, e.g., on the total time a test is allowed to execute
^ software systems metrics are expected to vary across multiple experiment iterations, thus it is important to repeat the experiment multiple times, to compute reliable performance metrics. More information can be found in section 4 of http://www.pautasso.info/biblio-pdf/benchflow-bpmds2017.pdf
+ More details about confidence intervals can be found on https://en.wikipedia.org/wiki/Confidence_interval and the referenced book: "A modern introduction to probability and statistics: understanding why and how"

The Quality Gates Specification

```
configuration:
 # OPTIONAL
 quality_gates:
   # OPTIONAL IF exploration_strategy.regression = mars
   mean_absolute_error: { String : <number><"%"> }
   # OPTIONAL IF goal.type = regression_complete OR goal.type =
    ↪  regression_intersection
   regression:
     # EITHER service name
     service: { String : <name of service> }
     # OR workload name
     workload: { String : <name of workload> }
     gate_metric: { String : <name of the workload or service metric> }
     # OPTIONAL
     regression_delta_absolute: { String : <amount><unit> }
     # OPTIONAL
     regression_delta_percent: { String : <number><"%"> }
   # OPTIONAL
   workloads:
     { String : <name of workload> }:
         # OPTIONAL
       - max_mix_deviation: { String : <number><"%"> }
         # OPTIONAL
         max_think_time_deviation: { String : <number><"%"> }
         # OPTIONAL
         gate_metric: { String : <name of the workload metric> }
```

```
26          # OPTIONAL
27          condition: { String : <a mathematical expression: >,<,>=,<=,=,+%,-%> }
28          # OPTIONAL
29          gate_threshold_target: { String : <a threshold value> OR Number : <value
             ↪  of the workload metric> }
30          # OPTIONAL
31          gate_threshold_minimum: { String : <a threshold value> OR Number : <value
             ↪  of the workload metric> }
32        ...
33      ...
34    # OPTIONAL
35    services:
36      { String : <name of service> }:
37        - gate_metric: { String : <name of the service metric> }
38          condition: { String : <a mathematical expression: >,<,>=,<=,=,+%,-%> }
39          gate_threshold_target: { String : <a threshold value> OR Number : <value
             ↪  of the service metric> }
40          # OPTIONAL
41          gate_threshold_minimum: { String : <a threshold value> OR Number : <value
             ↪  of the service metric> }
42        ...
43      ...
```

Listing D.6. Summative Evaluation: The Quality Gates YAML Format Specification

In Listing D.6, we report the specification of the `quality gates`. Users specify quality gates to define the outcome of the test according to different criteria, thus contributing to the goal-driven specification of the performance test. The outcome of the test at the end of its execution is usually `successful` or `failing`. The result is determined by evaluating every single gate and then applying the AND operator to the gates' results.
Quality gates can be specified on workloads (lines #18-#33) and service metrics (lines #35-#43). Gates are specified using dedicated metrics, such as `max_mix_deviation` (line #21), defining the maximum accepted deviation from the specified workload to interact with the SUT*, or `max_think_time_deviation` (line #23), defining the maximum accepted deviation in the time interleaving multiple operations interacting with the SUT with respect to the specified settings*. Conditions can also be specified on specific metrics (lines #24-#31 and #38-#41). For metrics, users can always report both a threshold indicating the wanted value for the metric, as well as a minimum accepted value. The gates are evaluated according to the minimum accepted

value. Reporting the expected threshold as well is important for better evaluating the test result (more details on https://www.stickyminds.com/article/better-way-reporting-performance-test-results). When relying on an exploration strategy building a prediction model, the user can also specify a quality gate to determine when the test can be considered successful according to the precision reached by the model (line #5).
For specific test types, such as the `regression`, is requested to specify customized quality gates. In this case, a specific section is provided as part of the language (lines #7-#16). We currently support a single regression condition based on an observed metric, applied either on a workload or a service. The condition checks for an absolute or a percentage variation among the versions specified as part of the regression test.
Full-size image: https://docs.google.com/drawings/d/13OIry1aP8t4FMqj8lbhFgamBkJMTU84VM4IeEik0_hI

* Refer to the next section for more details.

The Workloads, Operations, and their Mix Specification

```
1  workloads:
2   { String : <name of the workload> }
3     # OPTIONAL IF more than one workload
4     popularity: { String : <number><"%"> }
5     { String : <name of the workload item> }
6       driver_type: { String : <"start_bpmn" | "http"> }
7       # OPTIONAL IF goal.type = regression_complete OR goal.type =
        ↪  regression_intersection
8       sut_version: { String : <sut version> }
9       # OPTIONAL
10      target_service: { String : <name of service> } # default to
        ↪  default_target_service
11      # OPTIONAL
12      inter_operation_timings:  { String : <"negative_exponential" | "uniform" |
        ↪  "fixed_time"> }
13      # OPTIONAL IF more than one workload item
14      popularity: { String : <number><"%"> }
15      operations:
16        ...
17      # OPTIONAL IF driver_type = http (Web Services)
18      data_sources:
19        ...
```

```
    # OPTIONAL
    mix:
      ...
  ...
...
```

Listing D.7. Summative Evaluation: The Workload YAML Format Specification

```
workloads:
 { String : <name of the workload> }
   # OPTIONAL IF more than one workload
   popularity: { String : <number><"%"> }
   { String : <name of the workload item> }
     # IF driver_type = start_bpmn (WfMS)
     operations:
       - { String : <name of the .bpmn file with the process> }
       ...
     # IF driver_type = http (Web Services)
     operations:
       { String : <Name of operation> }
         protocol: { String : <"http" | "https"> }
         endpoint: { String : <path to call optionally referencing to data in
          ↪ body, parameter or extracted from response to other operations> }
         # REFERENCE:
          ↪ https://www.w3.org/Protocols/rfc2616/rfc2616-sec5.html#sec5.1.1
         method: { String : <"OPTIONS" | "GET" | "HEAD" | "POST" | "PUT" |
          ↪ "DELETE" | "TRACE" | "CONNECT"> }
         # OPTIONAL
         # REFERENCE: https://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html
         headers:
           Accept: { String : <Media Type> }
           ...
         # OPTIONAL
         query_parameter:
           - { String : <name of the query parameter> }:
             items:
               - { String : <content value> }
               ...
             # OPTIONAL
             retrieval: { String : <"sequential" | "random"> }
           ...
         # OPTIONAL
         url_parameter:
```

```
33            - { String : <name of the url parameter> }:
34              items:
35                - { String : <content value> }
36                ...
37              # OPTIONAL
38              retrieval: { String : <"sequential" | "random"> }
39            ...
40          # OPTIONAL
41          extract_regex:
42            - { String : <name assigned to the extracted value> }:
43              # REFERENCE: https://developer.mozilla
                ↪ .org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/match
44              pattern: { String : <a regex pattern to extract values> }
45              # OPTIONAL
46              default: { String : <the default value assigned when no values are
                ↪ extracted> }
47              # OPTIONAL
48              match_number: { Number : <the matched element index to select> }
49            ...
50          # OPTIONAL
51          extract_json:
52            - { String : <name assigned to the extracted value> }:
53              # REFERENCE: https://goessner.net/articles/JsonPath/
54              pattern: { String : <a JSON selector pattern to extract values> }
55              # OPTIONAL
56              default: { String : <the default value assigned when no values are
                ↪ extracted> }
57              # OPTIONAL
58              match_number: { Number : <the matched element index to select> }
59            ...
60          # OPTIONAL
61          body:
62            content:
63              - { String : <content value> }
64              ...
65            # OPTIONAL
66            retrieval: { String : <"sequential" | "random"> }
67          # OPTIONAL
68          body_file: { String : <name of a data_sources> }
69          # OPTIONAL
70          body_form:
71            - { String : <name of the form field> }:
72              items:
73                - { String : <content value> }
74                ...
75              # OPTIONAL
```

```
            retrieval: { String : <"sequential" | "random"> }
          ...
    # OPTIONAL IF driver_type = http (Web Services)
    data_sources:
      - path: { String : <path to a file containing data> }
        delimiter: { String : <delimiter used in the file containing data> }
        # OPTIONAL
        name: { String : <name assigned to the data source> }
        # OPTIONAL
        retrieval: { String : <"sequential" | "random"> }
      ...
    # OPTIONAL
    mix:
      ...
  ...
...
```

Listing D.8. Summative Evaluation: The Operations YAML Format Specification

```
workloads:
 { String : <name of the workload> }
   # OPTIONAL IF more than one workload
   popularity: { String : <number><"%"> }
   { String : <name of the workload item> }
     # OPTIONAL
     mix:
       # IF FixedSequenceMix
       fixed_sequence: { [String] : <name of operations in the wanted order> }
       # IF FlatMix. List of percentages (SUM = 100%) and optionally think time
       ↪  (tt), index refers to index of operation
       flat: { [String] : <number><"%"> <tt({ Number : <mean> } { Number :
       ↪  <deviation> })> }
       # IF FlatSequenceMix. List of percentages (SUM = 100%) and optionally think
       ↪  time (tt), index refers to index of sequence specified below
       flat: { [String] : <number><"%"> <tt({ Number : <mean> } { Number :
       ↪  <deviation> })> }
         sequences:
           - { [String] : <name of operations in the wanted order> }
           ...
       # IF MatrixMix (needs to be a square matrix). List of percentages (SUM =
       ↪  100%) and optionally think time (tt), index refers to index of
       ↪  operation
       matrix:
```

```
19          - { [String] : <number><"%"> <tt({ Number : <mean> } { Number :
            ↪  <deviation> })> }
20          ...
21    ...
22  ...
```

Listing D.9. Summative Evaluation: The Mix YAML Format Specification

In Listing D.7, Listing D.8, and Listing D.9, we report the specification of the `workloads`, the `operations`, and their `mix` respectively. We support workload specifications for both RESTful Web services (selecting `http` as `driver_type` on line #4 of the workloads specification) and BPMN20 WfMSs (selecting `start_bpmn` as `driver_type` on line #4 of the workloads specification).
The user can specify multiple `workloads`, representing different interaction scenarios with the SUT. The workload can be realized by different workload items (e.g., representing different scenarios of interaction with the SUT), target a specific SUT version (line #6 of the workloads specification), and can specify the target service (line #8 of the workloads specification). It is also possible to specify how to select the timings among the workload items operations (line #10 of the workloads specification)* and the `popularity` (line #12 of the workloads specification) of each workload item, useful to decide the weight when multiple workload items are specified.
As part of the workload, the user specifies `operations`. According to the `driver_type` the user has different ways to specify the operations (lines #4-#75 of the operations specification). Operations can extract data from SUT's response to be used in subsequent operations (lines #38-#57 of the operations specification) and can refer to `data_source` (lines #76-#84 of the operations specification), as a source of data for calls to the SUT (lines #20-#37 and #58-#75 of the operations specification). We omit some details of operation specifications because they basically refer to the HTTP standard. For more information, you can refer to https://www.w3.org/Protocols/rfc2616/rfc2616.html.
To decide how to mix the operations of a workload, the user specifies the `mix` (lines #5-#20 of the mix specification). The mix allows the definition of sequences of operations, or Markov chains^ controlling how to navigate from one operation to another one, as well as the time the simulated user spends to think about the next operation to execute. The framework executing the test attempts to respect as much as possible the specified mix of operations and think time. Due to the indeterminism of software system behavior, a

slight deviation from the same must be expected according to the known variance of SUT's behavior, the test execution environment, as well as the SUT deployment environments.
Full-size image for workloads: https://docs.google.com/drawings/d/1mQ5tfWB8APoxp_2T0gnwVvnWpBJtJxDjJr9eE_sfEwA
Full-size image for operation: https://docs.google.com/drawings/d/11DmtReRXdtB_qtoZNT8MKRTUPxE_5u6gXmsDKLla1dg
Full-size image for mix: https://docs.google.com/drawings/d/10SATky5NdwlzMsjC3pcLbFAokzvkfLwNSNhMIXEkjHA

* For more details about the timing among operations refer to section 7 on the following guide http://faban.org/1.3/docs/guide/driver/elements.html
^ For more information about the possible mixes, refer to section 6 of http://faban.org/1.3/docs/guide/driver/elements.html. For more details about Markov chain specification, please refer to https://www.se.informatik.uni-kiel.de/en/research/projects/markov4jmeter

The System Under Test Specification

```
sut:
 name: { String : <name of the sut> }
 versions:
   # EITHER specific values
   values: { [String] : <name of versions in the wanted order> }
   # OR range
   range: { [String, String] : <a list of versions (inclusive)> }
 # OPTIONAL
 type: { String : <"wfms" | "http"> }
 sut_configuration:
   default_target_service:
     name: { String : <name of target service> }
     endpoint: { String : <base endpoint for each operation> }
     # OPTIONAL
     sut_ready_log_check: { String : <a regular expression to check the
     ↪  availability of the sut> }
   # OPTIONAL
   deployment:
     { String : <name of service> }: { String : <name of server alias> }
     ...
 # OPTIONAL
```

```
21  services_configuration:
22    { String: <name of service> }:
23      # OPTIONAL
24      resources:
25        # OPTIONAL
26        cpu: { [String] : <a list of values + unit> }
27        # OPTIONAL
28        memory: { [String] : <a list of bytes value + unit> }
29      # OPTIONAL
30      configuration:
31        { String: <name of environment variable> }: { String: <value> }
32        ...
33    ...
```

Listing D.10. Summative Evaluation: The SUT YAML Format Specification

In Listing D.10, we report the specification of the system under test and its configuration (`sut`). The users can specify the `name` (line #2) of the SUT and the target `versions` (lines #3-#7), as well as the `type` (line #9).
The SUT has to be configured to automate the test (lines #10-#19). The user configures the default target service for the workload and optionally states how to identify when the SUT is ready to receive the load (line #15). The proposed framework can also take care of deploying the SUT for different experiments, thus the user can specify data about the deployment of different services (lines #17-#19). If no deployment information is provided, the proposed framework selects the best suitable and available machines where to deploy the SUT, unless a full `endpoint` is specified. When a full endpoint is specified, the SUT is expected to be already deployed at the specified endpoint, thus the deployment and the service configurations (lines #21-#33) are skipped*.
The user can also specify service configurations (lines #21-#33) in terms of service' resources and configuration variables.
Full-size image: https://docs.google.com/drawings/d/1DwL95XXfXilsSsCX68hS71q_VtCNrPxwHSSYAojCCIk

* Please note that in order for complex tests to be executed, the SUT deployment is expected to be carried out by the proposed framework, so that configurations requested for different experiments can be injected at deployment time.

The Data Collection Specification

```
1  # OPTIONAL
2  data_collection:
3   only_declared: { Boolean : <"true" | "false"> } # default "false"
4   # OPTIONAL
5   services:
6     # IF collector does NOT require CONFIGURATION
7     { String : <name of service> }: { String | [String] : <names of BenchFlow
       ↪  collectors> }
8     # IF some collectors requires CONFIGURATION
9     { String : <name of service> }:
10      { String : <name of BenchFlow collector> }:
11        configuration:
12          { String : <name of environment variable> }: { [String] : <a list of
             ↪  possible values> }
13          ...
14      ...
15    ...
16  # OPTIONAL
17  workloads:
18    { String : <name of the workload> }:
19      # OPTIONAL IF trial execution framework = JMeter
20      jmeter:
21        interval: { String : <time interval for data collection><"s" (seconds)> } #
           ↪  default 1s
22      # OPTIONAL IF trial execution framework = Faban
23      faban:
24        interval: { String : <time interval for data collection><"s" (seconds)> } #
           ↪  default 1s
25    ...
```

Listing D.11. Summative Evaluation: The Data Collection YAML Format Specification

In Listing D.11, we report the specification of the data collection services to be used to collect performance data used to compute the observed performance metrics.
Data collection services are available as part of the proposed solution and can collect client-side performance data (lines #17-#25), and server-side (lines #5-#15) resource utilization data, files produced by the SUT as well as dumps of databases (e.g., useful for computing BPMN20 WfMSs-specific metrics).

The services can be configured (lines #9-#14) according to SUT configurations (e.g., to properly connect to a DBMS) and the list of services can be extended according to new requirements. Values of configuration variables can refer to keywords representing variables injected by the framework part of the proposed approach at runtime, to refer to, e.g., IP addresses and Ports of services from which data have to be collected, or configurations already specified as part of the deployment descriptor part of the test bundle. Client-side collection services depend on the actual execution framework used to execute the experiments, and the user can specify the collection interval between two subsequent samplings of performance data. We currently support Faban (http://faban.org/) and JMeter (https://jmeter.apache.org/) as execution frameworks.
The proposed framework takes care of injecting collection services for the observed metrics, in case the user does not specify some of them and no custom configuration is required.
Full-size image: https://docs.google.com/drawings/d/1bJShriRttibBW8SmAStjxsBEFpOCHAYH4LIAniET3wE

Integrating Tests with a CSDL
The Test Suite Specification

```
1  version: { String : <Test Suite CDSL version, e.g. '1'> }
2  name: { String : <name of the performance test suite> }
3  # OPTIONAL
4  description: { String : <description of the performance test suite> }
5  suite:
6    triggers:
7      # OPTIONAL
8      scheduled: { Boolean : <"true" | "false"> } # default "false"
9      # OPTIONAL
10     on:
11       # OPTIONAL
12       push:
13         # OPTIONAL
14         branches:
15           - { String : <expression matching the name of branch> }
16           ...
17       # OPTIONAL
18       pull_request:
19         # OPTIONAL
20         contexts: { String : <"head" | "merge" | "all"> }
21         # OPTIONAL
```

```
22          source_branches:
23            - { String : <expression matching the name of branch> }
24            ...
25          # OPTIONAL
26          target_branches:
27            - { String : <expression matching the name of branch> }
28            ...
29        # OPTIONAL
30        releases:
31          types: { [String] : <types of releases> }
32        # OPTIONAL
33        deployments: { [String] : <names of deployments> }
34    # OPTIONAL IF DEPLOYMENTS
35    environments:
36      - { String : <name of environments> }:
37          # OPTIONAL
38          skip_deploy: { Boolean : <"true" | "false"> } # default "false"
39      ...
40    tests: -
41      # EITHER a list of file paths containing tests
42      - { String : <path of file containing test> }
43      ...
44      # OR
45      include_labels: { [String] : <expression matching the name of labels to select
        ↪  tests> }
46
47  quality_gates:
48    criterion: { String : <"all_success" | "at_least_one_success"> }
49    # OPTIONAL
50    exclude:
51      - { String : <path of file containing test> }
52      ...
```

Listing D.12. Summative Evaluation: The Tests Suite YAML Format Specification

In Listing D.12, we report the specification of the `test suite`. The user can specify test suites, to automatically select tests implemented using the specification presented above, in different moments of the CSDL lifecycle according to the generated events.
In CSDL software is usually committed to a versioning system by different users, and developed using multiple so-called feature branches. When features are considered mature enough, they are usually merged in a development version first and then in a production version of the SUT. These activities gen-

erate events such as `push`, `pull requests`, `releases` and `deployments`* (lines #10-#33), and these events are used to determine when to trigger different performance tests. In case of deployments, the user can also specify the environment of interest of such deployment to trigger the execution of performance tests (lines #35-#39) and indicate whether to skip the deployment of the SUT^ or not.
The selection of the tests realizing the test suite happens either using regular expressions matching `labels`, or by enumerating all the test specification files to execute (lines #40-#45). As per the test specification, also for test suites, the user is expected to define quality gates to determine the overall result of the test suite execution (lines #47-#52). The test suite can fail if at least one of the quality gates of the tests realizing the test suite fails (apart from the excluded ones as reported in lines #50-#52), or if none of the tests of the suite is successful. The second option is useful for example when testing different variants of the same SUT and it is sufficient that at least one is successful to proceed.
With the command line interface part of the proposed framework, the user can also submit information about the events generated as part of the CSDL, other than the tests and the test suites.
In case the test suite is not executed based on some events but scheduled to be executed, e.g., overnight, the user can specify so as well (line #8). The actual scheduling is expected to happen in the tools used for CSDL.
Full-size image: https://docs.google.com/drawings/d/1bW0cxksIjU6o-EEx7iZo0I-_t5mpj3bt2rGrvlfQO8E

* For more information about CSDL and the mentioned events refer to https://nvie.com/posts/a-successful-git-branching-model/
^ E.g., if it is already deployed by the CSDL, in such case the endpoint of the executed tests are expected to be specified and executed tests are not expected to requiring SUT configuration other than the default one
NOTE: termination criteria are not required for the integration with CSDL, because each single test part of the test suite is expected to define proper termination criteria.

D.4.2 Examples of Declarative Performance Tests Specification and Integration with CSDL

Example of a Load Test

```yaml
version: "3"
name: "Load Test"
description: "Example of Load Test"
labels: "load_test"
configuration:
  goal:
    type: "load_test"
    observe:
      workloads:
        workload_a: avg_response_time, avg_latency
        workload_b: avg_response_time, avg_latency
        workload_b.operation_a: avg_response_time
      services:
        service_a: avg_cpu, avg_memory
        service_b: avg_cpu, avg_memory
        dbms_a: avg_cpu, avg_memory, avg_io
  load_function:
    users: 1000
    ramp_up: 5m
    steady_state: 20m
    ramp_down: 5m
  termination_criteria:
    test:
      max_time: 40m
  quality_gates:
    workloads:
      workload_a:
        - max_mix_deviation: 5%
          max_think_time_deviation: 2%
          gate_metric: avg_response_time
          condition: "<="
          gate_threshold_target: "100ms"
          gate_threshold_minimum: "200ms"
sut:
  name: "my_app"
  versions:
    values: "v1.5"
  type: "http"
  sut_configuration:
    default_target_service:
      name: "service_a"
      endpoint: "/"
      sut_ready_log_check: "/(.*)System started(.*)/g"
    deployment:
      service_a: "my_server"
```

```yaml
  services_configuration:
    service_b:
      resources:
        cpu: 200m
        memory: 256Mi
      configuration:
        THREADPOOL_SIZE: 64
workloads:
  workload_a:
    item_a:
      driver_type: "http"
      inter_operation_timings: "negative_exponential"
      popularity: 80%
      operations:
        operation_a:
          protocol: "https"
          endpoint: "/"
          method: "GET"
          extract_regex:
            - title:
              pattern: "<title>(.*?)</title>"
              default: ""
              match_number: 1
        operation_b:
          protocol: "https"
          endpoint: "/${title}"
          method: "POST"
          body_file: "datasource_a"
        data_sources:
          - path: "/path_to_datasource_a"
            delimiter: ","
            name: datasource_a
            retrieval: "random"
        mix:
          max_deviation: 5%
          flat: "75.0% tt(1000.0 500.0), 25.0% tt(2000.0 400.0)"
    item_b:
      driver_type: "http"
      inter_operation_timings: "negative_exponential"
      popularity: 20%
      operations:
        operation_a:
          protocol: "https"
          endpoint: "/"
          method: "GET"
  workload_b:
```

```yaml
    item_a:
      driver_type: "http"
      target_service: "service_b"
      operations:
        operation_a:
          protocol: "https"
          endpoint: "/"
          method: "GET"
data_collection:
  # AUTOMATICALLY attached based on the observe section IF NOT specified
  services:
    service_a: "stats"
    service_b: "stats"
    dbms_a: "stats"
```

Listing D.13. Summative Evaluation: A Load Test for a Web service YAML Example

In Listing D.13, we report a complete example of a load test. The load test in the example states:

a) the load test goal, being the specified test a load test (line #7)

b) the metrics to observe, on workloads, operations, and services (lines #8-#16)

c) the target SUT and its configuration (lines #34-#52)

d) the load function (lines #17-#21)

e) the termination criterion for the test, based on maximum execution time (lines #22-#24)

f) the quality gates, referring to the workloads named “workload_a” and based on client-side performance metrics (lines #25-#33)

g) the two workloads, operations and mixes issued to the SUT and their popularity (lines #53-#99)

h) the data collection services, that in this case can also be automatically attached by the proposed framework (lines #100-#105)

The execution of such a load test is expected to last at most 40 minutes and compute the specified observed metrics.
Full-size image: https://docs.google.com/drawings/d/1vF8FaSNO4BR25yXSVrssVKo80lgTxbGPPvHn0KSVVsY

Example of a Smoke Test

```yaml
version: "3"
name: "Smoke Test"
description: "Example of Smoke Test"
labels: "smoke_test"
configuration:
  goal:
    type: "smoke_test"
    observe:
      ...
  load_function:
    users: 10
    ramp_up: 30s
    steady_state: 2m
    ramp_down: 30s
  termination_criteria:
    test:
      max_time: 20m
    ...
  quality_gates:
    ...
sut:
  ...
workloads:
  ...
data_collection:
  # AUTOMATICALLY attached based on the observe section IF NOT specified
  services:
    ...
```

Listing D.14. Summative Evaluation: A Smoke Test for a Web service YAML Example

In Listing D.14, we report an example of a smoke test. We omit many details in the specification, assuming they are the same as the previous load test. This is something we allow as part of the proposed solutions, to facilitate reusability of basic tests and opinionated templates defined by expert users and/or provided

as part of our approach. A user could define basic tests, including all the details about the workload, the SUT, collections services, etc and then override such specifications with an additional specification providing only the delta to be applied in addition (if not present in the basic specification) or in substitution (if present in the basic specification) information*.
In the provided example for the smoke test, assuming the user already defined the previous load test, she has only to specify:

a) the updated goal type to "smoke_test" (line #7)

b) the updated load function (lines #10-14) and termination criterion (lines #15-#17) because smoke tests usually need less time than load tests to be executed

The execution of such a smoke test is expected to last at most 20 minutes.
Full-size image: https://docs.google.com/drawings/d/1xgNGPG_6fgKKgDim1F2yxnC2M9FlyB7YUAkrxhKSSM0

* When submitting the test using the CLI provided as part of the approach, the user can specify multiple tests as part of the test bundle and they are merged overriding information from the first to the last in order of specification.

Example of Integration with CSDL

```
1  version: "1.6"
2  name: "Smoke Tests on Features"
3  description: "Runs all the Smoke Tests on new features merging to the Development
   ↪  branch"
4  suite:
5    triggers:
6      on:
7        pull_request:
8          contexts: "merge"
9          source_branches:
10           - "feat-*"
11         target_branches:
12           - "development"
13   tests:
14     include_labels: [ "(.*)smoke_test(.*)" ]
15
16 quality_gates:
```

```
17    criterion: "all_success"
18    exclude:
19      - "./tests/performance/smoke/smoke_test_alternative_scenario.yaml"
```

Listing D.15. Summative Evaluation: A Test Suite YAML Example

In Listing D.15, we report an example of test suite specification to integrate the execution of tests in CSDL. The proposed example refers to smoke tests the user wants to execute on developed feature branches. The test suite in the example states:

a) the triggers (lines #5-#12) defining the execution of the suite based on pull requests from feature branches to the development branch. The context indicated is "merge" meaning the test is requested to be executed on the version of the SUT built out of the feature branch codebase merged with the development codebase

b) the selection criteria of the tests to be included as part of the test suite (lines #13-#14). Selection criteria are based on the labels, selecting all the tests having as part of the labels the string "smoke_test" in order of file name under a "./test/performance" folder part of the same code repository where the SUT is developed. If no tests with the specified labels are found the test suite can not be executed

c) the quality gates (lines #16-#19) of the test suite. In the case of the example, the quality gates expect all the executed test to be successful, but the one reported in the `exclude` block (lines #18-#19)

The execution of such a test suite is expected to last until all the selected tests are executed.
Full-size image: https://docs.google.com/drawings/d/1eOlDELcfr5aEEX35KSO8JzYLOXWjoBKOs2Ug5Sl4B84

D.5 Review of the Approach

We are asking you to complete some tasks, to collect feedback about the learnability and reusability of the proposed approach

NOTE: you can go back to the previous section for reference if you

need to consult it during the following tasks. Going back to the previous page during the survey usually saves the already provided answers when coming back to the page containing the answer. If you think you might need to do this, we advise you to test the behavior with your browser to be sure about it, before completing multiple answers. For your convenience, we provide a PDF version of the "Overview of the Approach and Examples" so that you do not need to go back to that page: https://drive.google.com/file/d/1bvOW8QXQIfOwHbmDUEE2CGuOOfhi3Vfk

D.5.1 Learnability

In the following we are going to present you with a declarative performance test specification, or part of it, using the DSL proposed in the approach presented as part of this survey. After showing you the DSL, we ask you to pick one of the three options, representing the one you consider a correct statement according to what you learned about the DSL in the previous section.

Goal Specification

Full-size image: https://docs.google.com/drawings/d/1mSwLGBiwUj9foZtPczoudAVJ_4DUZdh8yYIlv5j5tKw

7a. What does the following DSL represents?

```yaml
version: "3"
name: "OMITTED"
description: "OMITTED"
labels: "OMITTED"
configuration:
  goal:
    type: "spike_test"
    observe:
      ...
  load_function:
    users: 5000
    ramp_up: 1m
    steady_state: 10s
    ramp_down: 1m
  termination_criteria:
    test:
      max_time: 20m
    ...
  quality_gates:
    ...
sut:
  ...
workloads:
  ...
data_collection:
  services:
    ...
```

- ☐ The user is specifying a load test, simulating 5000 users.
- ☐ The user is specifying a spike test, increasingly loading the SUT with more and more users for 1 minute until 5000 simulated users are reached.
- ☐ The user is specifying a spike test, increasingly loading the SUT with more and more users for 1 minute and 10 seconds until 5000 simulated users are reached.

7b. Indicate your considerations about the previous DSL element, if any:

_____

Exploration Space Specification

NOTE: "observe", "load_function", "termination_criteria" and "quality_gates" are OMITTED, but you can assume they are correctly specified. **Full-size image:** https://docs.google.com/drawings/d/1f0ECkQ7tqk5NBiq5DNTy9bqfS-DiB0onu1lu9WWcbwY

8a. What does the following DSL represents?

```
configuration:
  goal:
    type: "configuration"
    stored_knowledge: "true"
    observe:
      ... #OMITTED
    exploration:
      exploration_space:
        services:
          service_a:
            resources:
              cpu:
                range: [100m, 1000m]
                step: "*2"
              memory:
                range: [256Mi, 1024Mi]
                step: "+256Mi"
            configuration:
              NUM_SERVICE_THREAD: [12, 24]
          dbms_a:
            resources:
              cpu:
                range: [100m, 1000m]
                step: "*2"
              memory:
                range: [256Mi, 1024Mi]
                step: "+256Mi"
            configuration:
              QUERY_CACHE_SIZE: [48Mi, 64Mi, 88Mi, 112Mi]
      exploration_strategy:
        selection: "one_at_a_time"
  load_function:
    ... #OMITTED
  termination_criteria:
    ... #OMITTED
  quality_gates:
    ... #OMITTED
# sut, workloads, data_collection
```

- ☐ The specified configuration test configures the SUT to explore CPU, memory, and values for a configuration setting of "service_a" and "dbms_a" executing one experiment at a time. It does so by reusing results from previous executions of tests and experiments with the same specification.
- ☐ The specified configuration test configures the SUT to explore CPU, memory, and values for a configuration setting of "service_a" and "dbms_a" executing experiments by randomly selecting them. It does so by reusing results from previous executions of tests and experiments with the same specification.
- ☐ The specified configuration test is incomplete, ignoring the OMITTED specifications and the "sut", "workloads" and "data_collection" that can be assumed to be correctly defined.

8b. Indicate your considerations about the previous DSL element, if any:

Observe Specification

Full-size image: https://docs.google.com/drawings/d/1FnbD8LC_OWqemeMPjrABGfYHcQxYk7v74VAf8rFheH0

9a. What does the following DSL represents?

```
configuration:
  goal:
    ...
    observe:
      workloads:
        workload_a: avg_response_time, avg_latency
        workload_b: avg_response_time, avg_latency
        workload_b.operation_a: avg_response_time
      services:
        service_a: avg_cpu, avg_memory
        service_b: avg_cpu, avg_memory
        dbms_a: avg_cpu, avg_memory, avg_io
```

- ☐ In the reported "observe" section of a test specification the user is interested in both client-side and server-side performance data for different workloads and services part of the SUT respectively.
- ☐ In the reported "observe" section of a test specification the user is interested only in client-side performance data for different workloads.
- ☐ In the reported "observe" section of a test specification the user is interested only in server-side performance data for different services part of the SUT.

9b. Indicate your considerations about the previous DSL element, if any:

Termination Criteria Specification

Full-size image: https://docs.google.com/drawings/d/1rh6SWQ7zPStyV7rQKi7jWY6lxZfCzu16t8gG9QcNtl0

10a. What does the following DSL represents?

```yaml
configuration:
  termination_criteria:
    test:
      max_time: 120h
      max_failed_experiments: 10%
    experiment:
      workloads:
        workload_a:
          confidence_interval_metric: avg_response_time
          confidence_interval_value: 50ms
          confidence_interval_precision: 95%
      services:
        service_a:
          confidence_interval_metric: avg_cpu
          confidence_interval_value: 60%
          confidence_interval_precision: 95%
```

- ☐ In the reported "termination_criteria" section of a test specification the user wants to terminate the test after exactly 120 hours and when more than 10% of experiments are failed, or according to other conditions related to some metrics on a workload and a service.
- ☐ In the reported "termination_criteria" section of a test specification the user wants to terminate the test after at most 120 hours and when more than 10% of experiments are failed.
- ☐ In the reported "termination_criteria" section of a test specification the user wants to terminate the test after at most 120 hours or when more than 10% of experiments are failed, or according to other conditions related to some metrics on a workload and a service.

10b. Indicate your considerations about the previous DSL element, if any:

Quality Gates Specification

Full-size image: https://docs.google.com/drawings/d/1qBa0cN-sdVPEkZ3qwQRj2RczqKhVZv-8ITEeVM99hoE

11a. What does the following DSL represents?

```yaml
configuration:
  quality_gates:
    workloads:
      workload_a:
        - max_mix_deviation: 5%
          max_think_time_deviation: 2%
          gate_metric: avg_response_time
          condition: "<="
          gate_threshold_target: "100ms"
          gate_threshold_minimum: "200ms"
    services:
      service_a:
        - gate_metric: avg_cpu
          condition: "<="
          gate_threshold_target: 50%
          gate_threshold_minimum: 60%
```

- ☐ In the reported "quality_gates" section of a test specification the user considers the test successful when the "workload_a" has an average response time of less than or equal to 200 milliseconds or the average utilization of the CPU for the "service_a" is less than or equal to 60%.
- ☐ In the reported "quality_gates" section of a test specification the user considers the test successful when the "workload_a" has an average response time of less than or equal to 200 milliseconds and the average utilization of the CPU for the "service_a" is less than or equal to 60%.
- ☐ In the reported "quality_gates" section of a test specification the user considers the test successful when the "workload_a" has an average response time of less than or equal to 100 milliseconds and the average utilization of the CPU for the "service_a" is less than or equal to 50%.

11b. Indicate your considerations about the previous DSL element, if any:

Workloads Specification

Full-size image: https://docs.google.com/drawings/d/14NN-A648zeAU7Hu0FvG10XKHULTHbPaF-9ce0Lu4J9k

12a. What does the following DSL represents?

```
workloads:
  workload_a:
    item_a:
      driver_type: "http"
      inter_operation_timings: "negative_exponential"
      popularity: 80%
      operations:
        ...
    item_b:
      driver_type: "http"
      inter_operation_timings: "negative_exponential"
      popularity: 20%
      operations:
        ...
  workload_b:
    item_a:
      driver_type: "http"
      operations:
        ...
```

- ☐ In the reported "workloads" section of a test specification the user specifies two different HTTP workloads to simulate the interaction with the SUT and for the "workload_a" she models two workload items having a different probability of execution.

- ☐ In the reported "workloads" section of a test specification the user specifies two different HTTP workloads to simulate the interaction with the SUT and for the "workload_a" she models two workload items having the same probability of execution.

- ☐ In the reported "workloads" section of a test specification the user specifies two different HTTP workloads to simulate the interaction with the SUT and for the "workload_a" she models two workload items having a different probability of execution and for the "workload_b" she models a single workload item expecting to be executed 50% of the times.

12b. Indicate your considerations about the previous DSL element, if any:

_____

Operations Specification

Full-size image: https://docs.google.com/drawings/d/13JfsOnp_iF3IidS0fELzdjSOz5yBdgyI_t2mHqRfklA

13a. What does the following DSL represents?

```yaml
workloads:
  workload_a:
    item_a:
      ...
      operations:
        operation_a:
          protocol: "https"
          endpoint: "/"
          method: "GET"
          extract_regex:
            - title:
              pattern: "<title>(.*?)</title>"
              default: ""
              match_number: 1
        operation_b:
          protocol: "https"
          endpoint: "/${title}"
          method: "POST"
          body_file: "datasource_a"
        data_sources:
          - path: "/path_to_datasource_a"
            delimiter: ","
            name: datasource_a
            retrieval: "random"
        mix:
          ... #OMITTED
    item_b:
      ...
      operations:
        operation_a:
          protocol: "https"
          endpoint: "/"
          method: "GET"
  workload_b:
    item_a:
      driver_type: "http"
      target_service: "service_b"
      operations:
        operation_a:
          protocol: "https"
          endpoint: "/"
          method: "GET"
```

- In the reported "operations" section of a test specification the user defines two operations for the "workload_a". For "workload_b" she specifies a single operation, targeting "service_b". The operations are only using data retrieved from the response of previous operations.
- In the reported "operations" section of a test specification the user defines three operations for the "workload_a", two parts of the "item_a" and one part of the "item_b". For "workload_b" she specifies a single operation, targeting "service_b". Some operations are using data retrieved from the response of previous operations as well as data retrieved from a file.
- In the reported "operations" section of a test specification the user defines three operations for the "workload_a", two parts of the "item_a" and one part of the "item_b". For "workload_b" she specifies a single operation, targeting "service_b". The operations

are only using data retrieved from the response of previous operations.

13b. Indicate your considerations about the previous DSL element, if any:

Mix Specification

Full-size image: https://docs.google.com/drawings/d/157m9a4a2FU7u86fG_c5tPxeBFMK7975ck2MJgmffOSk

14a. What does the following DSL represents?

```
operations:
  operation_a:
    protocol: "https"
    endpoint: "/"
    method: "GET"
  operation_b:
    protocol: "https"
    endpoint: "/${title}"
    method: "POST"
    body_file: "datasource_a"
  data_sources:
    - path: "/path_to_datasource_a"
      delimiter: ","
      name: datasource_a
      retrieval: "random"
  mix:
      max_deviation: 5%
      flat: "75.0% tt(1000.0 500.0), 25.0% tt(2000.0 400.0)"
```

- ☐ In the reported "mix" section of a test specification the user expects the "operation_a" is chosen to be executed as the next operation 25% of the times, while "operation_b" 75% of the times. She also reports the expected think time of the simulated users before deciding which next operation to execute. The maximum accepted deviation from the specified mix is 5%.
- ☐ In the reported "mix" section of a test specification the user expects the "operation_a" is chosen to be executed as the next operation 75% of the times, while "operation_b" 25% of the times. The maximum accepted deviation from the specified mix is 5%.
- ☐ In the reported "mix" section of a test specification the user expects the "operation_a" is chosen to be executed as the next operation 75% of the times, while "operation_b" 25% of the times. She also reports the expected think time of the simulated users before deciding which next operation to execute. The maximum accepted deviation from the specified mix is 5%.

14b. Indicate your considerations about the previous DSL element, if any:

SUT Specification

Full-size image: https://docs.google.com/drawings/d/1wSM1FKcR_6SYLFd0fuAmWlHXl6YJ84fQT4z2FPslpRs

15a. What does the following DSL represents?

```
sut:
  name: "my_app"
  versions:
    values: "v1.5"
  type: "http"
  sut_configuration:
    default_target_service:
      name: "service_a"
      endpoint: "/"
      sut_ready_log_check: "/(.*)System started(.*)/g"
    deployment:
      service_a: "my_server"
  services_configuration:
    service_b:
      resources:
        cpu: 200m
        memory: 256Mi
      configuration:
        THREADPOOL_SIZE: 64
```

- ☐ In the reported "sut" section of a test specification the user is testing a SUT having at least two services and defines as default target for the operations the "service_a" deployed on "my_server" and specifies the configuration options for the "service_b" only in terms of resources.
- ☐ In the reported "sut" section of a test specification the user is testing a SUT having exactly two services and defines as default target for the operations the "service_a" deployed on "my_server" and specifies the configuration options for the "service_b" both in terms of resources and configuration variables.
- ☐ In the reported "sut" section of a test specification the user is testing a SUT having at least two services and defines as default target for the operations the "service_a" deployed on "my_server" and specifies the configuration options for the "service_b" both in terms of resources and configuration variables.

15b. Indicate your considerations about the previous DSL element, if any:

Data Collection Specification

Full-size image: https://docs.google.com/drawings/d/1VABQ0kphFQY9CG-xs7ACVVO3_tcCprUrWNuOgaP_5f8

16a. What does the following DSL represents?

```
data_collection:
  # AUTOMATICALLY attached based on the observe section IF NOT specified
  services:
    service_a: "stats"
    service_b: "stats"
    dbms_a:
      mysql:
        configuration:
          MYSQL_DB_NAME: database_name
          MYSQL_USER: mysql_user
          MYSQL_USER_PASSWORD: mysql_password
          TABLE_NAMES: TABLE_1, TABLE_2
          MYSQL_PORT: 3306
```

- ☐ In the reported "data_collection" section of a test specification the user is interested in collecting the "stats" (resource utilization) only for "service_a", and then she requests to store the data of "TABLE_1" and "TABLE_2" for the "dbms_a" that is expected to be a MySQL DBMS.
- ☐ In the reported "data_collection" section of a test specification the user is interested in collecting the "stats" (resource utilization) for "service_a" and "service_b", and then she requests to store the data of "TABLE_1" and "TABLE_2" for the "dbms_a" that is expected to be a MySQL DBMS.
- ☐ In the reported "data_collection" section of a test specification the user is interested in collecting the "stats" (resource utilization) for "service_a" and "service_b", and then she requests to store the data of "TABLE_1" for the "dbms_a" that is expected to be a MySQL DBMS.

16b. Indicate your considerations about the previous DSL element, if any:

Complete Test Specification - Regression Test

The following specification defines a regression test, checking for regression on all the operations of all the versions of the SUT part of the test. A regression test re-runs a defined workload under the same configurations to ensure that previously developed and tested features still perform the same after a change to the SUT. **Full-size image:** https://docs.google.com/drawings/d/1W3sPbaWOljbIIgbi3qJPC9vBW_SvVMZr5Ibb7feycxU

17a. What does the following DSL represents?

```
configuration:
  goal:
    type: "regression_complete" # OR regression_intersection
    stored_knowledge: "true"
    observe:
      workloads:
        workload_a: avg_response_time
        workload_b: avg_response_time
  load_function:
    users: 1000
    ramp_up: 5m
    steady_state: 20m
    ramp_down: 5m
  termination_criteria:
    test:
      max_time: 3h
    experiment:
      max_failed_trials: 0%
      workloads:
        workload_a:
          confidence_interval_metric: avg_response_time
          confidence_interval_value: 200ms
          confidence_interval_precision: 95%
  quality_gates:
    regression:
      workload: workload_a
      gate_metric: avg_response_time
      regression_delta_absolute: 50ms
    workloads:
      workload_a:
        - max_mix_deviation: 5%
          max_think_time_deviation: 2%
          gate_metric: avg_response_time
          condition: "<="
          gate_threshold_target: "150ms"
          gate_threshold_minimum: "250ms"
sut:
  name: "my_app"
  versions:
    values: [ "v1.5", "v1.6" ]
  type: "http"
  sut_configuration:
    ...
  services_configuration:
    service_a:
      resources:
        cpu: 500m
        memory: 512Mi
      configuration:
        THREADPOOL_SIZE: 64
```

- ☐ In the reported regression test specification the user is testing “v1.5” and “v1.6” of a SUT named “my_app” for regressions on the “workload_a” response time. The maximum accepted deviation from the average response time for the entire “workload_a” between the two versions is 50 milliseconds. For the test to be successful the “workload_a” average response time has to be less than or equal to 250 milliseconds for both the versions of the SUT. Better if it is less than or equal to 150 milliseconds.
- ☐ In the reported regression test specification the user is testing “v1.5” and “v1.6” of a SUT named “my_app” for regressions on the “workload_a” response time. The maximum accepted deviation from the average response time for the entire “workload_a” between the two versions is 150 milliseconds. For the test to be successful the “workload_a” average response time has to be less than or equal to 150 milliseconds for both the versions of the SUT. Better if it is less than

or equal to 250 milliseconds.

- ☐ In the reported regression test specification the user is testing "v1.5" and "v1.6" of a SUT named "my_app" for regressions on the "workload_a" response time. The maximum accepted deviation from the average response time for the entire "workload_a" between the two versions is 50 milliseconds. The test is even more successful if the average response time of the "workload_a" is less than or equal to 250 milliseconds for both the versions of the SUT.

17b. Indicate your considerations about the previous DSL element, if any:

Test Suite Specification

Full-size image: https://docs.google.com/drawings/d/1xI3xtF_0r47SUypkFM2GXoJ1hwKbQnT_U2XfsTCOZuU

18a. What does the following DSL represents?

```
version: "1.6"
name: "Load Tests on Development"
description: "Runs all the Load Tests on development code merging to the Production branch"
suite:
  triggers:
    on:
      pull_request:
        contexts: "merge"
        source_branches:
          - "development"
        target_branches:
          - "master"
  tests: -
    include_labels: [ "(.*)load_test(.*)" ]

quality_gates:
  criterion: "all_success"
  exclude:
    - "./tests/performance/load/load_test_alternative_scenario.yaml"
```

- ☐ In the reported test suite specification the user wants to execute smoke tests when the Development branch is merged on the Production branch (master). The suite is successful if all the executed tests are successful, but the one stored in the "load_test_alternative_scenario.yaml" file.
- ☐ In the reported test suite specification the user wants to execute load tests when the Development branch is merged on the Production branch (master). The suite is successful if all the executed tests are successful, but the one stored in the "load_test_alternative_scenario.yaml" file.
- ☐ In the reported test suite specification the user wants to execute load tests when the Development branch is merged on the Production branch (master). The suite is successful when at least one of the

executed tests is successful, excluding the one stored in the "load_-test_alternative_scenario.yaml" file.

18b. Indicate your considerations about the previous DSL element, if any:

D.6 Reusability

In the following we are going to present you with multiple declarative performance test specifications, or part of them, using the DSL proposed in the approach presented as part of this survey. After showing you the DSL, we ask you to pick one of the three options, representing the one you consider the correct answer to the question we propose. The objective is to evaluate the possibility to reuse already defined specifications, for new goals a user might have.

Goal Specification Reuse

Full-size image: https://docs.google.com/drawings/d/11LDGWd7kR6L2PWcvu2qyyx8NtIa_5KNYzAbPbfkpAiA

19. An acceptance test is a test conducted to determine if the requirements of a specification or contract are met. Which of the following specifications would you reuse for defining an acceptance test issuing a load of 1000 simulated users for 20 minutes, adding only quality gates expressing the acceptance criteria, assuming termination criteria are already correctly specified?

☐ Option 1

```
configuration:
  goal:
    type: "load_test"
    observe:
      ...
  load_function:
    users: 1000
    ramp_up: 5m
    steady_state: 20m
    ramp_down: 5m
  termination_criteria:
    ...
```

☐ Option 2

```yaml
configuration:
  goal:
    type: "regression_complete"
    stored_knowledge: "true"
    observe:
      ...
  load_function:
    users: 1000
    ramp_up: 5m
    steady_state: 20m
    ramp_down: 5m
  termination_criteria:
    ...
```

☐ Option 3

```yaml
configuration:
  goal:
    type: "configuration"
    stored_knowledge: "true"
    observe:
      ... #OMITTED
    exploration:
      exploration_space:
        services:
          service_a:
            resources:
              cpu:
                range: [100m, 1000m]
                step: "*2"
              memory:
                range: [256Mi, 1024Mi]
                step: "+256Mi"
            configuration:
              NUM_SERVICE_THREAD: [12, 24]
      exploration_strategy:
        selection: "one_at_a_time"
```

Termination Criteria Specification Reuse

Full-size image: https://docs.google.com/drawings/d/13lcrUZn3qgz6CIZb2W0pFydIjX7BHSr8R4N1U6GAalc

20. Which of the following specifications would you reuse for defining termination criteria allowing the test to execute for a maximum of 24 hours, and iterating the experiment execution until the average response time of the workload is within a specified confidence interval with the given precision?

☐ Option 1

```yaml
termination_criteria:
  test:
    max_time: 24h
  experiment:
    services:
      service_a:
        confidence_interval_metric: avg_cpu
        confidence_interval_value: 60%
        confidence_interval_precision: 95%
```

☐ Option 2

```yaml
termination_criteria:
  test:
    max_time: 24h
  experiment:
    max_failed_trials: 10%
    workloads:
      workload_a:
        confidence_interval_metric: avg_response_time
        confidence_interval_value: 200ms
        confidence_interval_precision: 95%
```

☐ Option 3

```yaml
termination_criteria:
  test:
    max_failed_experiments: 0%
  experiment:
    max_failed_trials: 10%
    workloads:
      workload_a:
        confidence_interval_metric: avg_response_time
        confidence_interval_value: 200ms
        confidence_interval_precision: 95%
```

Quality Gate Specification Reuse

Full-size image: https://docs.google.com/drawings/d/1DL6knjXlU4mub7W9B5sDkIj-TDZYd-cWbYJSPE04tQE

21. Which of the following specifications would you reuse for defining quality gates declaring the test as successful if the average response time of the workload is below or equal to 200 milliseconds, and the time spent by simulated users in thinking before executing the next operation is within a 2% deviation from the specified one?

☐ Option 1

```yaml
quality_gates:
  workloads:
    workload_a:
      - max_mix_deviation: 5%
        max_think_time_deviation: 2%
        gate_metric: avg_response_time
        condition: "<="
        gate_threshold_target: "200ms"
        gate_threshold_minimum: "300ms"
```

☐ Option 2

```
quality_gates:
  workloads:
    workload_a:
      - max_mix_deviation: 5%
        max_think_time_deviation: 5%
        gate_metric: avg_response_time
        condition: "<="
        gate_threshold_target: "100ms"
        gate_threshold_minimum: "250ms"
```

☐ Option 3

```
quality_gates:
  workloads:
    workload_a:
      - max_think_time_deviation: 2%
        gate_metric: avg_response_time
        condition: "<="
        gate_threshold_target: "100ms"
        gate_threshold_minimum: "200ms"
```

Test Suite Specification Reuse

Full-size image: https://docs.google.com/drawings/d/1DVCoA6gr77G5tq6QlFvUT2qqwY5T5E07Hri-aoX3BII

22. Which of the following specifications would you reuse for defining a test suite of load tests executed on a pull request merging a feature to the development branch, and checking all the executed tests are successful to declare the entire test suite as successful?

☐ Option 1

```
version: "1.6"
name: "OMITTED"
description: "OMITTED"
suite:
  triggers:
    on:
      pull_request:
        contexts: "merge"
        source_branches:
          - "feature-*"
        target_branches:
          - "development"
  tests:
    include_labels: [ "(.*)load_test(.*)" ]

quality_gates:
  criterion: "all_success"
```

☐ Option 2

```yaml
version: "1.6"
name: "OMITTED"
description: "OMITTED"
suite:
  triggers:
    on:
      pull_request:
        contexts: "merge"
        source_branches:
          - "feature-*"
        target_branches:
          - "development"
  tests:
    include_labels: [ "(.*)load_test(.*)", "(.*)smoke_tests(.*)" ]

quality_gates:
  criterion: "all_success"
  exclude:
    - "./tests/performance/load/load_test_alternative_scenario.yaml"
```

☐ Option 3

```yaml
version: "1.6"
name: "OMITTED"
description: "OMITTED"
suite:
  triggers:
    on:
      pull_request:
        contexts: "merge"
        source_branches:
          - "development"
        target_branches:
          - "master"
  tests:
    include_labels: [ "(.*)load_test(.*)" ]

quality_gates:
  criterion: "all_success"
```

D.7 Final Evaluations

D.7.1 Expressiveness

23a. How do you evaluate the overall expressiveness of the proposed DSL?

When referring to the context of performance testing definition and automation, particularly in CSDL, report how you evaluate the overall expressiveness of the DSL proposed in this survey. Very Poor ☐—☐—☐—☐—☐ Excellent

23b. How do you motivate the previous answer about expressiveness?:

_____

D.7.2 Usability

24a. How useful do you consider the proposed approach for the target users of the same?

When referring to the target users of our approach, namely developers, software testers, quality assurance engineers, and operations engineers, report how useful you consider the proposed approach for them. Not usable ☐—☐—☐—☐—☐ Very usable

24b. How do you motivate the previous answer about expressiveness?:

25a. How useful do you consider the goal-based specification for the target users of the same?
When referring to the target users of our approach, namely developers, software testers, quality assurance engineers, and operations engineers, report how useful you consider the goal-based specification for them. Not usable ☐—☐—☐—☐—☐ Very usable

25b. How do you motivate the previous answers about the usability of the goal-based specification for the target users?:

D.7.3 Others

26a. How much more/less do you think the proposed approach would help the target users of the same in implementing and executing performance tests, compared to the standard imperative approaches?
Report how much more or less the proposed approach would help the target users in implementing and executing performance tests compared to standard imperative approaches, such for example the one proposed by JMeter. Much less ☐—☐—☐—☐—☐ Much more

26b. How do you motivate the previous answers about the suitability of the proposed approach for the target users?:

D.7.4 Generic

27. Would you be interested in using the framework implementing the proposed approach?

☐ Yes.

☐ No.

☐ Maybe.

28. What are the main PROs of the proposed approach?:
State what are the main PROs of the proposed approach according to your opinion

---

29. What are the main CONs of the proposed approach?:
State what are the main CONs of the proposed approach according to your opinion

---

30. Name similar approaches you are aware of:
Name similar approaches you are aware of. Providing a reference to the literature and/or a Website would be helpful

---

D.8 Thank you!

As anticipated at the beginning of the survey, we value the time you spent on providing the feedback, so we want to give you back something to show our appreciation.

We are giving away three little gifts as a sign of appreciation, raffled among all the participants completing the survey. The three gifts are all the same, the value is € 20 each, and at most one can be assigned to each participant. They are going to be discount coupons for well-known websites providing services and products.

The raffle is going to happen by the middle of August 2020, and only if at least 40 complete surveys are submitted*. To increase the probability of the raffle to happens, please share the survey with people you know that might contribute with valuable answers according to the defined target audience for this survey. Sharing the survey is very helpful for us.

The raffle is going to be executed on `https://commentpicker.com/random-name-picker.php` (with anonymized emails) so that we can provide you back with proof of results.

Leave your email address if you want to participate in the raffle. We

are going to write you back when the raffle results are available, and if you are one of the lucky winners, we are going to attach to the email the gift as well.

* You were able to check the current number of submitted responses by clicking on "See previous responses" on the next screen after clicking on the "Submit" button. I had to disable this because responses might contain sensitive data, such as the email address. Status at 31.07.2020: 63 responses.

31. What is your email address?:

_____

References

[Abedi and Brecht, 2017] Abedi, A. and Brecht, T. (2017). Conducting Repeatable Experiments in Highly Variable Cloud Computing Environments. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering (ICPE)*, pages 287–292.

[Ahmed et al., 2016] Ahmed, T. M., Bezemer, C.-P., Chen, T.-H., Hassan, A. E., and Shang, W. (2016). Studying the effectiveness of application performance management (APM) tools for detecting performance regressions for web applications. In *Proceedings of the 13th ACM International Workshop on Mining Software Repositories (MSR)*, pages 1–12.

[Akamai, 2020] Akamai (2020). Akamai - Web and Mobile Performance. https://www.akamai.com/us/en/products/performance/. Last visited on February 7, 2021.

[Aleti et al., 2018] Aleti, A., Trubiani, C., van Hoorn, A., and Jamshidi, P. (2018). An efficient method for uncertainty propagation in robust software performance estimation. *Journal of Systems and Software*, 138(1):222–235.

[AlGhamdi et al., 2020] AlGhamdi, H. M., Bezemer, C.-P., Shang, W., Hassan, A. E., and Flora, P. (2020). Towards reducing the time needed for load testing. *Journal of Software: Evolution and Process*, N/D(N/D).

[Alhaj, 2014] Alhaj, M. (2014). *Automatic Derivation of Performance Models in the Context of Model-Driven SOA*. PhD thesis, Ottawa Carleton Institute of Electrical and Computer Engineering.

[Almeida et al., 2004] Almeida, V. A. F., Dowdy, L. W., and Menascé, D. A. (2004). *Performance by design: computer capacity planning by example*. Prentice Hall.

[Apache Software Foundation, 2020a] Apache Software Foundation (2020a). Apache Cassandra. https://cassandra.apache.org/. Last visited on February 7, 2021.

[Apache Software Foundation, 2020b] Apache Software Foundation (2020b). Apache Kafka. https://kafka.apache.org/. Last visited on February 7, 2021.

[Apache Software Foundation, 2020c] Apache Software Foundation (2020c). Apache Spark - Unified Analytics Engine for Big Data. https://spark.apache.org/. Last visited on February 7, 2021.

[Apica, 2016] Apica (2016). Website Testing, Optimization and Monitoring. https://www.apicasystem.com. Last visited on February 7, 2021.

[Apte et al., 2017] Apte, V., Viswanath, T. V. S., Gawali, D., Kommireddy, A., and Gupta, A. (2017). AutoPerf: Automated load testing and resource usage profiling of multi-tier internet applications. In *Proceedings of the 8th ACM/SPEC International Conference on Performance Engineering (ICPE)*, pages 115–126.

[Arcelli et al., 2015] Arcelli, D., Cortellessa, V., and Trubiani, C. (2015). Performance-Based Software Model Refactoring in Fuzzy Contexts. In *Proceedings of the 18th International Conference on Fundamental Approaches to Software Engineering (FASE)*, pages 149–164.

[Ardagna et al., 2010] Ardagna, D., Tanelli, M., Lovera, M., and Zhang, L. (2010). Black-box performance models for virtualized web service applications. In *Proceedings of the 21th Symposium on Software Performance (SOSP)*, pages 153–164.

[Arif et al., 2018] Arif, M. M., Shang, W., and Shihab, E. (2018). Empirical study on the discrepancy between performance testing results from virtual and physical environments. *Empirical Software Engineering*, 23(3):1490–1518.

[Artillery, 2014] Artillery (2014). Artillery. https://artillery.io/. Last visited on February 7, 2021.

[Avritzer et al., 2018] Avritzer, A., Ferme, V., Janes, A., Russo, B., Schulz, H., and van Hoorn, A. (2018). A Quantitative Approach for the Assessment of Microservice Architecture Deployment Alternatives by Automated Performance Testing. In *Proceedings of the 10th European Conference on Software Architecture (ECSA)*, pages 159–174.

[Avritzer et al., 2020] Avritzer, A., Ferme, V., Janes, A., Russo, B., van Hoorn, A., Schulz, H., Menasché, D., and Rufino, V. Q. (2020). Scalability Assessment of Microservice Architecture Deployment Configurations: A Domain-based Approach Leveraging Operational Profiles and Load Tests. *Journal of Systems and Software*, 165(1):110564–16.

[Avritzer et al., 2019] Avritzer, A., Menasché, D. S., Rufino, V. Q., Russo, B., Janes, A., Ferme, V., van Hoorn, A., and Schulz, H. (2019). PPTAM - Production and Performance Testing Based Application Monitoring. In *Proceedings of the 10th ACM/SPEC International Conference on Performance Engineering Companion (ICPE Companion)*, pages 39–40.

[Bachiega et al., 2018] Bachiega, N. G., Souza, P. S. L., Bruschi, S. M., and de Souza, S. d. R. S. (2018). Container-Based Performance Evaluation: A Survey and Challenges. In *Proceedings of the 2018 IEEE International Conference on Cloud Engineering (IC2E)*, pages 398–403.

[Balsamo et al., 2004] Balsamo, S., Simeoni, M., Inverardi, P., and Di Marco, A. (2004). Model-based performance prediction in software development: a survey. *IEEE Transactions on Software Engineering*, 30(5):295–310.

[Barik et al., 2016] Barik, R. K., Lenka, R. K., Rao, K. R., and Ghose, D. (2016). Performance analysis of virtual machines and containers in cloud computing. In *Proceedings of the 2nd International Conference on Computing, Communication and Automation (ICCCA)*, pages 1204–1210.

[Barna et al., 2011] Barna, C., Litoiu, M., and Ghanbari, H. (2011). Autonomic load-testing framework. In *Proceedings of the 8th ACM International Conference on Autonomic Computing (ICAC)*, pages 91–100.

[Ben-Ari and Steinberg, 2007] Ben-Ari, E. N. and Steinberg, D. M. (2007). Modeling Data from Computer Experiments: An Empirical Comparison of Kriging with MARS and Projection Pursuit Regression. *Quality Engineering*, 19(4):327–338.

[BenchExec, 2015] BenchExec (2015). BenchExec: A Framework for Reliable Benchmarking and Resource Measurement. https://github.com/sosy-lab/benchexec. Last visited on February 7, 2021.

[Bernardino et al., 2016] Bernardino, M., Rodrigues, E. M., and Zorzo, A. F. (2016). Performance testing modeling. In *Proceedings of the 31st ACM Symposium on Applied Computing (SAC)*, pages 1660–1665.

[Bernardino et al., 2014] Bernardino, M., Zorzo, A. F., and Rodrigues, E. M. (2014). Canopus: A Domain-Specific Language for Modeling Performance Testing. In *Proceedings of the 9th IARIA International Conference on Software Engineering Advances (ICSEA)*, pages 157–167.

[Beyer et al., 2016] Beyer, B., Jones, C., Petoff, J., and Murphy, N. R. (2016). *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media, Inc., 1st edition.

[Bezemer et al., 2019] Bezemer, C.-P., Eismann, S., Ferme, V., Grohmann, J., Heinrich, R., Jamshidi, P., Shang, W., van Hoorn, A., Villavicencio, M., Walter, J., and Willnecker, F. (2019). How is Performance Addressed in DevOps? In *Proceedings of the 10th ACM/SPEC International Conference on Performance Engineering (ICPE)*, pages 45–50.

[Bianculli et al., 2010a] Bianculli, D., Binder, W., and Drago, M. L. (2010a). Automated performance assessment for service-oriented middleware: a case study on BPEL engines. In *Proceedings of the 19th ACM International Conference on World Wide Web (WWW)*, pages 141–150.

[Bianculli et al., 2010b] Bianculli, D., Binder, W., and Drago, M. L. (2010b). SOABench - performance evaluation of service-oriented middleware made easy. In *Proceedings of the 32nd IEEE International Conference on Software Engineering (ICSE)*, pages 301–302.

[Biswas et al., 2011] Biswas, S., Mall, R., Satpathy, M., and Sukumaran, S. (2011). Regression Test Selection Techniques - A Survey. *Informatica*, 35(3):289–322.

[Blackburn et al., 2006] Blackburn, S. M., Garner, R., Hoffmann, C., Khan, A. M., McKinley, K. S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S. Z., Hirzel, M., Hosking, A. L., Jump, M., Lee, H. B., Moss, J. E. B., Phansalkar, A., Stefanovic, D., VanDrunen, T., von Dincklage, D., and Wiedermann, B. (2006). The DaCapo benchmarks - java benchmarking development and analysis. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications (OOPSLA)*, pages 169–190.

[Blackfire, 2014] Blackfire (2014). Blackfire. https://blackfire.io. Last visited on February 7, 2021.

[Blazemeter, 2014] Blazemeter (2014). Taurus: Automation-friendly framework for Continuous Testing. https://gettaurus.org/. Last visited on February 7, 2021.

[Blazemeter, 2016] Blazemeter (2016). BlazeMeter. https://www.blazemeter.com. Last visited on February 7, 2021.

[Blohm et al., 2016] Blohm, M., Vogel, S., Pahlberg, M., and Walter, J. (2016). Kieker4DQL: Declarative Performance Measurement. In *Proceedings of the 27th Symposium on Software Performance (SOSP)*, pages 1–3.

[Bondi, 2014] Bondi, A. B. (2014). *Foundations of Software and System Performance Engineering: Process, Performance Modeling, Requirements, Testing, Scalability, and Practice*. Addison-Wesley Professional.

[Bosch, 2014] Bosch, J. (2014). *Continuous Software Engineering*. Springer.

[Boudec, 2011] Boudec, J.-Y. L. (2011). *Performance Evaluation of Computer and Communication Systems*. EFPL Press.

[Brad Kemp, 2001] Brad Kemp (2001). PET Scanner Performance: Quality Assurance and Acceptance Testing. https://www.aapm.org/meetings/05AM/pdf/18-2633-72827-103.pdf. Last visited on February 7, 2021.

[Brebner, 2016] Brebner, P. (2016). Automatic Performance Modelling from Application Performance Management (APM) Data - An Experience Report. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering (ICPE)*, pages 55–61.

[Brebner et al., 2005] Brebner, P., Cecchet, E., Marguerite, J., Tůma, P., Ciuhandu, O., Dufour, B., Eeckhout, L., Frénot, S., Krishna, A. S., Murphy, J., and Verbrugge, C. (2005). Middleware benchmarking: approaches, results, experiences. *Concurrency and Computation: Practice and Experience*, 17(15):1799–1805.

[Brown and Hellerstein, 2004] Brown, A. B. and Hellerstein, J. L. (2004). An approach to benchmarking configuration complexity. In *Proceedings of the 11th ACM SIGOPS European Workshop (EW)*, pages 1–5.

[Bruel et al., 2019] Bruel, J.-M., Mazzara, M., and Meyer, B., editors (2019). *DevOps Round-Trip Engineering: Traceability from Dev to Ops and Back Again*.

[Brunnert and Krcmar, 2017] Brunnert, A. and Krcmar, H. (2017). Continuous performance evaluation and capacity planning using resource profiles for enterprise applications. *Journal of Systems and Software*, 123(1):239–262.

[Brunnert et al., 2015] Brunnert, A., van Hoorn, A., Willnecker, F., Danciu, A., Hasselbring, W., Heger, C., Herbst, N. R., Jamshidi, P., Jung, R., von Kistowski, J., Koziolek, A., Kroß, J., Spinner, S., Vögele, C., Walter, J., and Wert, A. (2015). Performance-oriented DevOps: A Research Agenda. Technical report, SPEC Research Group - DevOps Performance Working Group.

[Bulej, 2016] Bulej, L. (2016). Performance Testing in Software Development - Getting the Developers on Board. In *Proceedings of the 9th ACM/SPEC International Conference on Performance Engineering Companion (ICPE Companion)*, pages 9–9.

[Bulej et al., 2014] Bulej, L., Bures, T., Horký, V., Kotrč, J., Marek, L., and Trojanek, T. (2014). SPL: Unit Testing Performance. Technical report, Charles University.

[Bulej et al., 2005] Bulej, L., Kalibera, T., and Tůma, P. (2005). Repeated results analysis for middleware regression benchmarking. *Performance Evaluation*, 60(1-4):345–358.

[Burton et al., 1990] Burton, S. J., Sudweeks, R. R., Merrill, P. F., and Wood, B. (1990). *How to prepare better multiple-choice test items: Guidelines for university faculty*. PhD thesis, Brigham Young University. Department of Instructional Science.

[Canfora and Di Penta, 2009] Canfora, G. and Di Penta, M. (2009). Service-Oriented Architectures Testing: A Survey. In De Lucia, A. and Ferrucci, F., editors, *Software Engineering: International Summer Schools on Software Engineering*, pages 78–105. Springer.

[Casale, 2016] Casale, G. (2016). Performance Aware DevOps. http://www.hit.bme.hu/~buttyan/atnc/20160122_Casale_Devops.pdf. Last visited on February 7, 2021.

[Cerfoglio, 2016] Cerfoglio, G. (2016). *The BenchFlow framework for flexible performance data collection and analysis*. Master's thesis, Università della Svizzera italiana.

[Chen et al., 2017] Chen, T., Syer, M. D., Shang, W., Jiang, Z. M., Hassan, A. E., Nasser, M., and Flora, P. (2017). Analytics-Driven Load Testing: An Industrial Experience Report on Load Testing of Large-Scale Systems. In *Proceedings of the 39th International Conference on Software*

Engineering: Software Engineering in Practice Track (ICSE SEIP), pages 243–252.

[Chhetri et al., 2016] Chhetri, M. B., Chichin, S., Vo, Q. B., and Kowalczyk, R. (2016). Smart CloudBench - A framework for evaluating cloud infrastructure performance. *Information Systems Frontiers*, 18(3.):413–428.

[Cito et al., 2016] Cito, J., Ferme, V., and Gall, H. C. (2016). Using Docker Containers to Improve Reproducibility in Software and Web Engineering Research. In *Proceedings of the 16th International Conference on Web Engineering (ICWE)*, pages 609–612.

[Clemson, Toby, 2014] Clemson, Toby (2014). Testing Strategies in a Microservice Architecture. Technical report, ThoughtWorks.

[CodeShip, 2016] CodeShip (2016). Continuous Integration Essentials. https://codeship.com/continuous-integration-essentials. Last visited on February 7, 2021.

[Cortellessa et al., 2011] Cortellessa, V., Di Marco, A., and Inverardi, P. (2011). *Model-Based Software Performance Analysis*. Springer.

[Costa et al., 2020] Costa, V., Girardon, G., Bernardino, M., Machado, R., Legramante, G., Neto, A., Basso, F. P., and de Macedo Rodrigues, E. (2020). Taxonomy of Performance Testing Tools: A Systematic Literature Review. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing (SAC)*, pages 1997–2004.

[Crolotte, 2009] Crolotte, A. (2009). Issues in Benchmark Metric Selection. In *Proceedings of the 1st Technology Conference on Performance Evaluation and Benchmarking (TPCTC)*, pages 146–152.

[Cunha et al., 2013] Cunha, M., Mendonça, N. C., and Sampaio, A. (2013). A Declarative Environment for Automatic Performance Evaluation in IaaS Clouds. In *Proceedings of the 6th IEEE International Conference on Cloud Computing (CLOUD)*, pages 285–292.

[CustomerCentrix LLC, 2020] CustomerCentrix LLC (2020). LoadStorm - Cloud Load Testing Tool. https://loadstorm.com/. Last visited on February 7, 2021.

[D'Ambrogio et al., 2001] D'Ambrogio, A., Iazeolla, G., and Cortellessa, V. (2001). Automatic derivation of software performance models from CASE documents. *Performance Evaluation*, 45(2-3):81–105.

[D'Avico, 2016] D'Avico, S. (2016). *The BenchFlow framework for automated performance experiments execution on heterogeneous middleware systems*. Master's thesis, Università della Svizzera italiana.

[Di Alesio et al., 2013] Di Alesio, S., Nejati, S., Gotlieb, A., and Briand, L. (2013). Stress testing of task deadlines - A constraint programming approach. In *Proceedings of the 24th IEEE International Symposium on Software Reliability Engineering (ISSRE)*, pages 158–167.

[Ding, 2019] Ding, Z. (2019). Towards the Use of the Readily Available Tests from the Release Pipeline as Performance Tests. Are We There Yet? Master's thesis, Concordia University.

[Dlugi et al., 2015] Dlugi, M., Brunnert, A., and Krcmar, H. (2015). Model-based performance evaluations in continuous delivery pipelines. In *Proceedings of the 1st ACM International Workshop on Quality-aware Devops (QUDOS)*, pages 25–26.

[Docker Inc., 2013] Docker Inc. (2013). Docker. https://www.docker.com. Last visited on February 7, 2021.

[Docker Inc., 2015] Docker Inc. (2015). Docker Compose. https://docs.docker.com/compose/. Last visited on February 7, 2021.

[Driessen, 2010] Driessen, V. (2010). A successful Git branching model. https://nvie.com/posts/a-successful-git-branching-model/. Last visited on February 7, 2021.

[Duvall et al., 2007] Duvall, P., Matyas, S. M., and Glover, A. (2007). *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley Signature Series. Addison-Wesley.

[Dynamics, 2014] Dynamics, G. (2014). Jagger. https://griddynamics.github.io/jagger/doc/index.html. Last visited on February 7, 2021.

[Dynatrace, 2016] Dynatrace (2016). Dynatrace's Load. https://www.dynatrace.com/capabilities/load-testing/. Last visited on February 7, 2021.

[Dynatrace, 2019] Dynatrace (2019). Keptn. https://keptn.sh/. Last visited on February 7, 2021.

[Eden, 2011] Eden, M. (2011). A Survey of Performance Modeling and Analysis. In *Proceedings of the 1st ACM International Conference on Performance Engineering (WOSP/SIPEW)*, pages 153–163.

[Eismann et al., 2020] Eismann, S., Bezemer, C.-P., Shang, W., Okanović, D., and van Hoorn, A. (2020). Microservices: A Performance Tester's Dream or Nightmare? In *Proceedings of the 11th ACM/SPEC International Conference on Performance Engineering (ICPE)*, pages 138–149.

[ElasTest, 2016] ElasTest (2016). ElasTest: An elastic platform to ease end to end testing. https://elastest.eu/. Last visited on February 7, 2021.

[Elbaum et al., 2014] Elbaum, S. G., Rothermel, G., and Penix, J. (2014). Techniques for improving regression testing in continuous integration development environments. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 235–245.

[Erich et al., 2017] Erich, F., Amrit, C., and Daneva, M. (2017). A qualitative study of DevOps usage in practice. *Journal of Software: Evolution and Process*, 29(6):1–20.

[Evans, 2001] Evans, C. C. (2001). YAML. http://www.yaml.org. Last visited on February 7, 2021.

[Faban, 2000] Faban (2000). Faban. http://faban.org. Last visited on February 7, 2021.

[Farcic, 2016] Farcic, V. (2016). *The DevOps 2.0 Toolkit*. Packt Publishing.

[Ferme et al., 2015] Ferme, V., Ivanchikj, A., and Pautasso, C. (2015). A Framework for Benchmarking BPMN 2.0 Workflow Management Systems. In *Proceedings of the 13th Business Process Management Conference (BPM)*, pages 251–259.

[Ferme et al., 2016a] Ferme, V., Ivanchikj, A., and Pautasso, C. (2016a). Estimating the Cost for Executing Business Processes in the Cloud. In *Proceedings of the 14th Business Process Management Conference (BPM)*, pages 72–88.

[Ferme et al., 2016b] Ferme, V., Ivanchikj, A., Pautasso, C., Skouradaki, M., and Leymann, F. (2016b). A Container-centric Methodology for Benchmarking Workflow Management Systems. In *Proceedings of the 6th International Conference on Cloud Computing and Services Science (CLOSER)*, pages 74–84.

[Ferme et al., 2019] Ferme, V., Ivanchikj, A., Pautasso, C., Skouradaki, M., and Leymann, F. (2019). IT-Centric Process Automation: Study About the Performance of BPMN 2.0 Engines. In Lübke, D. and Pautasso, C., editors, *Empirical Studies on the Development of Executable Business Processes*, pages 167–197. Springer.

[Ferme et al., 2017a] Ferme, V., Lenhard, J., Harrer, S., Geiger, M., and Pautasso, C. (2017a). Workflow management systems benchmarking - unfulfilled expectations and lessons learned. In *Proceedings of the 38th IEEE International Conference on Software Engineering Companion (ICSE Companion)*, pages 379–381.

[Ferme and Pautasso, 2016] Ferme, V. and Pautasso, C. (2016). Integrating Faban with Docker for Performance Benchmarking. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering (ICPE)*, pages 129–130.

[Ferme and Pautasso, 2017] Ferme, V. and Pautasso, C. (2017). Towards Holistic Continuous Software Performance Assessment. In *Proceedings of the 9th ACM/SPEC International Conference on Performance Engineering Companion (ICPE Companion)*, pages 159–164.

[Ferme and Pautasso, 2018] Ferme, V. and Pautasso, C. (2018). A Declarative Approach for Performance Tests Execution in Continuous Software Development Environments. In *Proceedings of the 9th ACM/SPEC International Conference on Performance Engineering (ICPE)*, pages 261–272.

[Ferme et al., 2017b] Ferme, V., Skouradaki, M., Ivanchikj, A., Pautasso, C., and Leymann, F. (2017b). Performance Comparison Between BPMN 2.0 Workflow Management Systems Versions. In *Proceedings of the 18th Enterprise, Business-Process and Information Systems Modeling (BPMDS)*, pages 1–15.

[Findahl, 2017] Findahl, J. (2017). Automating goal-driven performance tests in BenchFlow. Master's thesis, Università della Svizzera italiana.

[Fitzgerald and Stol, 2017] Fitzgerald, B. and Stol, K.-J. (2017). Continuous software engineering: A roadmap and agenda. *Journal of Systems and Software*, 123(1):176–189.

[Forsgren et al., 2018] Forsgren, N., Humble, J., and Kim, G. (2018). *Accelerate: The Science of Lean Software and DevOps Building and Scaling High Performing Technology Organizations*. IT Revolution Press, 1st edition.

[Fowler, 2010] Fowler, M. (2010). *Domain-Specific Languages*. Pearson Education.

[Frank, 2013] Frank, U. (2013). Domain-Specific Modeling Languages - Requirements Analysis and Design Guidelines. In *Domain Engineering*, pages 133–157. Springer.

[Fredrich, Todd, 2012] Fredrich, Todd (2012). REST API Tutorial - Resource Naming. http://www.restapitutorial.com/lessons/restfulresourcenaming.html. Last visited on February 7, 2021.

[Gambi et al., 2016] Gambi, A., Pezzè, M., and Toffetti, G. (2016). Kriging-Based Self-Adaptive Cloud Controllers. *IEEE Transactions on Services Computing*, 9(3):368–381.

[Gatling, 2011] Gatling (2011). Gatling. http://gatling.io. Last visited on February 7, 2021.

[Gazzola et al., 2020] Gazzola, L., Goldstein, M., Mariani, L., Segall, I., and Ussi, L. (2020). Automatic Ex-Vivo Regression Testing of Microservices. In *Proceedings of the 1st International Conference on Automation of Software Test (AST)*, pages 11–20.

[Geiger et al., 2014] Geiger, C., Przytarski, D., and Thullner, S. (2014). Performance testing in continuous integration environments. Technical report, University of Stuttgart.

[Gerostathopoulos et al., 2016] Gerostathopoulos, I., Bures, T., Schmid, S., Horký, V., Prehofer, C., and Tůma, P. (2016). Towards Systematic Live Experimentation in Software-Intensive Systems of Systems. In *Proceedings of the 10th European Conference on Software Architecture (ECSA)*, pages 1–7.

[Gillmann et al., 2000] Gillmann, M., Mindermann, R., and Weikum, G. (2000). Benchmarking and Configuration of Workflow Management Systems. In Scheuermann, P. and Etzion, O., editors, *Proceedings of the 7th International Conference on Cooperative Information Systems (CoopIS)*, pages 186–197.

[GitLab, 2020] GitLab (2020). GitLab - Load Performance Testing. https://about.gitlab.com/. Last visited on February 7, 2021.

[Golovin et al., 2017] Golovin, D., Solnik, B., Moitra, S., Kochanski, G., Karro, J., and Sculley, D. (2017). Google Vizier - A Service for Black-Box Optimization. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 1487–1495.

[Google, 2019] Google (2019). 2019 Accelerate State of DevOps Report. Technical report, Dora.

[Google Cloud Platform, 2020] Google Cloud Platform (2020). PerfKit Benchmarker. https://github.com/GoogleCloudPlatform/PerfKitBenchmarker. Last visited on February 7, 2021.

[Goparaju et al., 2012] Goparaju, S. P., Ayesha, F., and Sanghamitra, P. (2012). Measuring the Influence of Process Automation on the Productivity of Software Development Teams. Master's thesis, Universiteit van Amsterdam.

[Gottesheim, 2015] Gottesheim, W. (2015). Challenges, Benefits and Best Practices of Performance Focused DevOps. In *Proceedings of the 4th ACM International Workshop on Large-Scale Testing (LT)*, pages 3–3.

[Grambow et al., 2019] Grambow, M., Lehmann, F., and Bermbach, D. (2019). Continuous Benchmarking: Using System Benchmarking in Build Pipelines. In *Proceedings of the 2019 IEEE International Conference on Cloud Engineering (IC2E)*, pages 241–246.

[Grano et al., 2019] Grano, G., Laaber, C., Panichella, A., and Panichella, S. (2019). Testing with Fewer Resources: An Adaptive Approach to Performance-Aware Test Case Generation. *IEEE Transactions on Software Engineering*, N/D(N/D):1–16.

[Gray, 1992] Gray, J. (1992). *The Benchmark Handbook for Database and Transaction Systems*. Morgan Kaufmann, 2nd edition.

[Guru99, 2014] Guru99 (2014). Sanity Testing Vs Smoke Testing: Introduction & Differences. https://www.guru99.com/smoke-sanity-testing.html. Last visited on February 7, 2021.

[Guru99, 2014] Guru99 (2014). Software Testing Tutorial: Free Course. https://www.guru99.com/software-testing.html. Last visited on February 7, 2021.

[Hackernoon, 2020] Hackernoon (2020). The State of Containers in 2020. Technical report, Hackernoon.

[Haghighatkhah et al., 2017] Haghighatkhah, A., Lwakatare, L. E., and Rodríguez, P. (2017). Continuous deployment of software intensive products and services: A systematic mapping study. *Journal of Systems and Software*, 123(1):263–291.

[Harrer et al., 2017] Harrer, S., Lenhard, J., Kopp, O., Ferme, V., and Pautasso, C. (2017). A Pattern Language for Workflow Engine Conformance and Performance Benchmarking. In *Proceedings of the European Conference on Pattern Languages of Programs (EuroPLoP)*, pages 1–46.

[Hashemian et al., 2017] Hashemian, R., Carlsson, N., Krishnamurthy, D., and Arlitt, M. (2017). IRIS: Iterative and Intelligent Experiment Selection. In *Proceedings of the 8th ACM/SPEC International Conference on Performance Engineering (ICPE)*, pages 143–154.

[Hauck et al., 2013] Hauck, M., Kuperberg, M., Huber, N., and Reussner, R. H. (2013). Ginpex: Deriving Performance-relevant Infrastructure Properties Through Goal-oriented Experiments. In *Proceedings of the 7th ACM SIGSOFT Symposium Quality of Software Architectures and Architecting Critical Systems (QoSA+ISARCS)*, pages 1345–1365.

[Helali Moghadam et al., 2019] Helali Moghadam, M., Saadatmand, M., Borg, M., Bohlin, M., and Lisper, B. (2019). Machine Learning to Guide Performance Testing: An Autonomous Test Framework. In *Proceedings of the 2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 164–167.

[Herbold, 2015] Herbold, S. (2015). How fast will your application go? Static and dynamic techniques for application performance modeling. https://htor.inf.ethz.ch/publications/index.php?pub=218. Last visited on February 7, 2021.

[Herbst et al., 2013] Herbst, N. R., Kounev, S., and Reussner, R. H. (2013). Elasticity in Cloud Computing: What It Is, and What It Is Not. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC)*, pages 23–27.

[Herbst et al., 2015] Herbst, N. R., Kounev, S., Weber, A., and Groenda, H. (2015). BUNGEE: An Elasticity Benchmark for Self-Adaptive IaaS Cloud Environments. In *Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 46–56.

[Hilton et al., 2016] Hilton, M., Tunnell, T., Huang, K., Marinov, D., and Dig, D. (2016). Usage, costs, and benefits of continuous integration in open-source projects. In *ASE'16: ACM/IEEE International Conference on Automated Software Engineering*, pages 426–437.

[Hoefler and Belli, 2015] Hoefler, T. and Belli, R. (2015). Scientific Benchmarking of Parallel Computing Systems: Twelve Ways to Tell the Masses When Reporting Performance Results. In *Proceedings of the 27th International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–12.

[Hollingsworth, 1995] Hollingsworth, D. (1995). The Workflow Reference Model. Technical report, Workflow Management Coalition Specification.

[Horký et al., 2013] Horký, V., Haas, F., Kotrč, J., Lacina, M., and Tůma, P. (2013). Performance Regression Unit Testing: A Case Study. In *Proceedings of the 10th European Workshop on Performance Engineering (EPEW)*, pages 149–163.

[Horký et al., 2016] Horký, V., Kotrč, J., Libič, P., and Tůma, P. (2016). Analysis of Overhead in Dynamic Java Performance Monitoring. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering (ICPE)*, pages 275–286.

[Horký et al., 2015] Horký, V., Libič, P., Marek, L., Steinhauser, A., and Tůma, P. (2015). Utilizing Performance Unit Tests To Increase Performance Awareness. In *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering (ICPE)*, pages 289–300.

[HP, 2016] HP (2016). HP LoadRunner. http://www8.hp.com/it/it/software-solutions/loadrunner-load-testing/. Last visited on February 7, 2021.

[Huck et al., 2008] Huck, K. A., Hernandez, O., Bui, V., Chandrasekaran, S., Chapman, B., Malony, A. D., McInnes, L. C., and Norris, B. (2008). Capturing performance knowledge for automated analysis. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing (SP)*, pages 1–10.

[Huebner et al., 2000] Huebner, F., Meier-Hellstern, K. S., and Reeser, P. (2000). Performance Testing for IP Services and Systems. In *Proceedings of the 1st German Workshop on Performance Engineering within Software Development (WOSP)*, pages 283–299.

[Humble and Farley, 2010] Humble, J. and Farley, D. (2010). *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley.

[Huppler, 2009] Huppler, K. (2009). The Art of Building a Good Benchmark. In *Performance Evaluation and Benchmarking*, pages 18–30. Springer.

[IBM, 2020] IBM (2020). IBM Rational Performance Tester. https://www.ibm.com/products/ibm-rational-performance-tester. Last visited on February 7, 2021.

[IBM Developer, 2019] IBM Developer (2019). YAML basics in Kubernetes. https://developer.ibm.com/technologies/containers/tutorials/yaml-basics-and-usage-in-kubernetes/. Last visited on February 7, 2021.

[ISO Central Secretary, 2009] ISO Central Secretary (2009). Quantities and units - Part 1: General. Technical report, International Organization for Standardization.

[Itkonen et al., 2017] Itkonen, J., Lassenius, C., and Laukkanen, E. (2017). Problems, causes and solutions when adopting continuous delivery - A systematic literature review. *Information and Software Technology*, 82(2):55–79.

[Ivanchikj et al., 2015] Ivanchikj, A., Ferme, V., and Pautasso, C. (2015). BPMeter - Web Service and Application for Static Analysis of BPMN 2.0 Collections. In *Proceedings of the 13th Business Process Management Conference (BPM)*, pages 30–34.

[Ivanchikj et al., 2017] Ivanchikj, A., Ferme, V., and Pautasso, C. (2017). On the Performance Overhead of BPMN Modeling Practices. In *Proceedings of the 15th Business Process Management Conference (BPM)*, pages 1–15.

[Jabbari et al., 2016] Jabbari, R., bin Ali, N., Petersen, K., and Tanveer, B. (2016). What is DevOps?: A Systematic Mapping Study on Definitions and Practices. In *Proceedings of the 16th ACM Scientific Workshop Proceedings of XP (XP Workshops)*, pages 12–11.

[Jaewon et al., 2018] Jaewon, L., Changkyu, K., Kun, L., Liqun, C., Govindaraju, R., and Jangwoo, K. (2018). WSMeter. *ACM SIGPLAN Notices*, 53(2):549–563.

[Jayasinghe et al., 2014] Jayasinghe, D., Malkowski, S., Li, J., Wang, Q., Wang, Z., and Pu, C. (2014). Variations in Performance and Scalability: An Experimental Study in IaaS Clouds Using Multi-Tier Workloads. *IEEE Transactions on Services Computing*, 7(2):293–306.

[Jiang and Hassan, 2015] Jiang, Z. M. and Hassan, A. E. (2015). A Survey on Load Testing of Large-Scale Software Systems. *IEEE Transactions on Software Engineering*, 41(11):1091–1118.

[Jimenez et al., 2016] Jimenez, I., Maltzahn, C., Lofstead, J., Moody, A., Mohror, K., Arpaci-Dusseau, R., and Arpaci-Dusseau, A. (2016). Characterizing and Reducing Cross-Platform Performance Variability Using OS-Level Virtualization. In *Proceedings of the 30th IEEE International Parallel and Distributed Processing Symposium Workshop (IPDPSW)*, pages 1077–1080.

[JMeter, 1998] JMeter (1998). JMeter. http://jmeter.apache.org. Last visited on February 7, 2021.

[Johnson et al., 2007] Johnson, M. J., Ho, C.-W., Maximilien, E. M., and Williams, L. (2007). Incorporating Performance Testing in Test-Driven Development. *IEEE Software*, 24(3):67–73.

[Kalibera and Tůma, 2006] Kalibera, T. and Tůma, P. (2006). Precise Regression Benchmarking with Random Effects: Improving Mono Benchmark Results. In *Proceedings of the 3rd European Workshop on Performance Engineering (EPEW)*, pages 63–77.

[Karsai et al., 2014] Karsai, G., Krahn, H., Pinkernell, C., Rumpe, B., Schindler, M., and Völkel, S. (2014). Design Guidelines for Domain Specific Languages. In *Proceedings of the 29th Workshop on Domain-Specific Modeling (OOPSLA)*, pages 1–10.

[Kempf et al., 2008] Kempf, T., Karuri, K., and Gao, L. (2008). *Software Instrumentation*. John Wiley & Sons.

[Klinaku and Ferme, 2018] Klinaku, F. and Ferme, V. (2018). Towards Generating Elastic Microservices: A Declarative Specification for Consistent Elasticity Configurations. In *Proceedings of the 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 510–513.

[Kolb et al., 2006] Kolb, R., Ganesan, D., Muthig, D., Kagino, M., and Teranishi, H. (2006). Goal-Oriented Performance Analysis of Reusable Software Components. In *Proceedings of the 9th International Conference on Software Reuse (ICSR)*, pages 368–381.

[Kounev et al., 2020] Kounev, S., Lange, K.-D., and von Kistowski, J. (2020). *Systems Benchmarking: For Scientists and Engineers*. Springer.

[Kowall, 2016] Kowall, J. (2016). The Role of APM in Continuous Integration and Continuous Release. https://blog.appdynamics.com/engineering/the-role-of-apm-in-continuous-integration-and-continuous-release/. Last visited on February 7, 2021.

[Kroß et al., 2016] Kroß, J., Willnecker, F., Zwickl, T., and Krcmar, H. (2016). PET: continuous performance evaluation tool. In *Proceedings of the 2nd ACM International Workshop on Quality-aware Devops (QUDOS)*, pages 42–43.

[Kubernetes, 2014] Kubernetes (2014). Kubernetes. https://kubernetes.io. Last visited on February 7, 2021.

[Kuperberg, 2010] Kuperberg, M. (2010). *Quantifying and predicting the influence of execution platform on software component performance.* PhD thesis, Karlsruhe Institute of Technology.

[Lahmar et al., 2015] Lahmar, I. b., Belhaj, N., and Mohamed, M. (2015). Collaborative Autonomic Container for the Management of Component-based Applications. In *Proceedings of the 24th IEEE International Conference on enabling Technologies (WETICE)*, pages 41–43.

[Lämmel, 2018] Lämmel, R. (2018). The Notion of a Software Language. In *Software Languages*, pages 1–49. Springer.

[Leitner and Bezemer, 2017] Leitner, P. and Bezemer, C.-P. (2017). An Exploratory Study of the State of Practice of Performance Testing in Java-Based Open Source Projects. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering (ICPE)*, pages 373–384.

[Leitner and Cito, 2016] Leitner, P. and Cito, J. (2016). Patterns in the Chaos - A Study of Performance Variation and Predictability in Public IaaS Clouds. *ACM Transactions on Internet Technology*, 16(3):1–23.

[Lenhard et al., 2017] Lenhard, J., Ferme, V., Harrer, S., Geiger, M., and Pautasso, C. (2017). Lessons Learned from Evaluating Workflow Management Systems. *ICSOC Workshops*, 10797(3):215–227.

[Levy and Steinberg, 2011] Levy, S. and Steinberg, D. M. (2011). Computer experiments: a review. *Advances in Statistical Analysis*, 94(4):311–324.

[Lewis and Fowler, 2014] Lewis, J. and Fowler, M. (2014). Microservices. https://martinfowler.com/articles/microservices.html. Last visited on February 7, 2021.

[Leymann and Roller, 2000] Leymann, F. and Roller, D. (2000). *Production Workflow: Concepts and Techniques*. Prentice Hall PTR.

[Li et al., 2009] Li, X., Huai, J., Liu, X., Zeng, J., and Huang, Z. (2009). SOArMetrics - A Toolkit for Testing and Evaluating SOA Middleware. In *Proceedings of the 1st IEEE World Conference on Services (SERVICES)*, pages 1–10.

[Li et al., 2016] Li, Y., Gupta, Y., Miller, E. L., and Long, D. D. E. (2016). Pilot: A Framework that Understands How to Do Performance Benchmarks the Right Way. In *Proceedings of the 24th IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 169–178.

[Likert, 1932] Likert, R. (1932). A technique for the measurement of attitudes. *Archives of Psychology*, 140(55).

[Little, 1961] Little, J. D. C. (1961). A Proof for the Queuing Formula: L = λW. *Operations Research*, 9(3):383–387.

[Liu, 2011] Liu, H. H. (2011). *Software Performance and Scalability.* A Quantitative Approach. John Wiley & Sons.

[Lizhi et al., 2020] Lizhi, L., Jinfu, C., Heng, L., Yi, Z., Weiyi, S., Jianmei, G., Catalin, S., Andrei, T., and Sarah, S. (2020). Using Black-Box Performance Models to Detect Performance Regressions under Varying Workloads: An Empirical Study. *Empirical Software Engineering*, 25(5):1–36.

[Lloyd et al., 2014] Lloyd, J. R., Duvenaud, D., Grosse, R., Tenenbaum, J. B., and Ghahramani, Z. (2014). Automatic Construction and Natural-Language Description of Nonparametric Regression Models. In *Proceedings of the 28th AAAI Conference on Artificial Intelligence (AAAI)*, pages 1242–1250.

[Load Impact AB, 2019] Load Impact AB (2019). K6. https://k6.io/. Last visited on February 7, 2021.

[Locust, 2020] Locust (2020). Locust - An open source load testing tool. https://locust.io/. Last visited on February 7, 2021.

[Logz.io, 2016] Logz.io (2016). DevOps Pulse. Technical report, Logz.io.

[Lwakatare et al., 2015] Lwakatare, L. E., Kuvaja, P., and Oivo, M. (2015). Dimensions of DevOps. In *Proceedings of the 15th ACM Scientific Workshop Proceedings of XP (XP Workshops)*, pages 212–217.

[Mani et al., 2013] Mani, N., Petriu, D. C., and Woodside, M. (2013). Propagation of incremental changes to performance model due to SOA design pattern application. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering (ICPE)*, pages 89–100.

[Marin et al., 2014] Marin, G., Dongarra, J., and Terpstra, D. (2014). MIAMI: A framework for application performance diagnosis. In *Proceedings of the 15th IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 158–168.

[Marin and Mellor-Crummey, 2004] Marin, G. and Mellor-Crummey, J. (2004). Cross-architecture performance predictions for scientific applications using parameterized models. *ACM SIGMETRICS Performance Evaluation Review*, 32(1):2–13.

[Markov, 1906] Markov, A. A. (1906). Extension of the law of large numbers to dependent quantities (in Russian). *Bulletin of the Society of the Physics Mathematics*, 15(2):135–156.

[Memon et al., 2017] Memon, A., Gao, Z., Nguyen, B., Dhanda, S., Nickell, E., Siemborski, R., and Micco, J. (2017). Taming Google-scale Continuous Testing. In *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE SEIP)*, pages 233–242.

[Meszaros, 2007] Meszaros, G. (2007). *xUnit Test Patterns*. Refactoring Test Code. Pearson Education.

[Micallef and Colombo, 2015] Micallef, M. and Colombo, C. (2015). Lessons learnt from using DSLs for automated software testing. In *Proceedings of the 8th IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 1–6.

[Michael et al., 2017] Michael, N., Ramannavar, N., Shen, Y., Patil, S., and Sung, J.-L. (2017). CloudPerf - A Performance Test Framework for Distributed and Dynamic Multi-Tenant Environments. In *Proceedings of the 8th ACM/SPEC International Conference on Performance Engineering (ICPE)*, pages 189–200.

[MicroFocus, 2020] MicroFocus (2020). Silk Performer - Realistic, Powerful Load and Stress Testing. https://www.microfocus.com/en-us/products/silk-performer/overview. Last visited on February 7, 2021.

[MinIO, 2020] MinIO (2020). MinIO | High Performance, Kubernetes Native Object Storage. https://min.io/. Last visited on February 7, 2021.

[Molyneaux, 2014] Molyneaux, I. (2014). *The Art of Application Performance Testing*. From Strategy to Tools. O'Reilly Media.

[Montgomery and Runger, 2013] Montgomery, D. C. and Runger, G. C. (2013). *Applied Statistics and Probability for Engineers*. John Wiley & Sons, 6 edition.

[Mostafa et al., 2017] Mostafa, S., Wang, X., and Xie, T. (2017). PerfRanker: Prioritization of Performance Regression Tests for Collection-Intensive Software. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 23–34.

[Mühlbauer et al., 2020] Mühlbauer, S., Apel, S., and Siegmund, N. (2020). Identifying Software Performance Changes Across Variants and Versions. In *Proceedings of the 35th IEEE/ACM Automated Software Engineering (ASE)*, pages 1–12.

[Murty, 2016] Murty, S. (2016). Performance Testing in a Nutshell. https://www.thoughtworks.com/insights/blog/performance-testing-nutshell. Last visited on February 7, 2021.

[Nambiar et al., 2016] Nambiar, M., Kattepur, A., Bhaskaran, G., Singhal, R., and Duttagupta, S. (2016). Model Driven Software Performance Engineering: Current Challenges and Way Ahead. *ACM SIGMETRICS Performance Evaluation Review*, 43(4):53–62.

[NeoLoad, 2016] NeoLoad (2016). NeoLoad | Automated Performance Testing Tool. https://www.neotys.com/neoload/overview. Last visited on February 7, 2021.

[Neotys, 2018] Neotys (2018). A Practical Guide to Performance Testing. https://www.neotys.com/wp-content/uploads/2018/01/WP__A_Practical_Guide_to_Performance_Testing-EN.pdf. Last visited on February 7, 2021.

[Nicolas Niclausse, 2017] Nicolas Niclausse (2017). Tsung - A distributed Load Testing. http://tsung.erlang-projects.org/. Last visited on February 7, 2021.

[North, 2006] North, D. (2006). Introducing BDD. http://dannorth.net/introducing-bdd/. Last visited on February 7, 2021.

[Okanović et al., 2020] Okanović, D., Beck, S., Merz, L., Zorn, C., Merino, L., van Hoorn, A., and Beck, F. (2020). Can a Chatbot Support Software Engineers with Load Testing? Approach and Experiences. In *Proceedings of the 11th ACM/SPEC International Conference on Performance Engineering (ICPE)*, pages 120–129.

[Okanović et al., 2019] Okanović, D., van Hoorn, A., Zorn, C., Beck, F., Ferme, V., and Walter, J. (2019). Concern-Driven Reporting of Software Performance Analysis Results. In *Proceedings of the Companion of the 2019 ACM/SPEC International Conference on Performance Engineering (ICPE)*, pages 1–4.

[opsZero, LLC, 2016] opsZero, LLC (2016). opsZero. https://www.acksin.com. Last visited on February 7, 2021.

[Pachidi and Spruit, 2015] Pachidi, S. and Spruit, M. R. (2015). The Performance Mining Method - Extracting Performance Knowledge from Software Operation Data. *International Journal of Business Intelligence Research*, 6(1):11–29.

[Palenga, 2018] Palenga, M. (2018). Declarative user experience regression analysis in continuous performance engineering. Master's thesis, University of Stuttgart.

[Pautasso et al., 2015] Pautasso, C., Leymann, F., Ferme, V., Skouradaki, M., and Roller, D. H. (2015). Towards Workflow Benchmarking - Open Research Challenges. In *Proceedings of the 16th Conference on Database Systems for Business, Technology, and Web of the German Informatics Society (BTW)*, pages 1–20.

[PayU, 2020] PayU (2020). Predator - ruthless api performance testing. https://www.predator.dev/. Last visited on February 7, 2021.

[PerfCake, 2016] PerfCake (2016). PerfCake. https://www.perfcake.org. Last visited on February 7, 2021.

[Petkovich et al., 2015] Petkovich, J. C., Oliveira, A., Zhang, Y., Reidemeister, T., and Fischmeister, S. (2015). DataMill: a distributed heterogeneous infrastructure for robust experimentation. *Software: Practice and Experience*, 46(10):1411–1440.

[Philip Aston, 2014] Philip Aston (2014). The Grinder, a Java Load Testing Framework. http://grinder.sourceforge.net/. Last visited on February 7, 2021.

[Pitakrat and Heinish, 2016] Pitakrat, T. and Heinish, J. (2016). A Reference Platform for Software Performance Engineering in DevOps. In *Proceedings of the AWS Summit*, pages 1–20.

[Podelko, 2008] Podelko, A. (2008). Agile Performance Testing. In *Proceedings of the 34th International Computer Measurement Group Conference (CMG)*, pages 267–278.

[Podelko, 2016] Podelko, A. (2016). Reinventing performance testing. In *Proceedings of the imPACt*, pages N–D. CMG.

[PractiTest, 2020] PractiTest (2020). State of Testing Report. https://www.practitest.com/resource/state-of-testing-report-2020/. Last visited on February 7, 2021.

[Puppet, 2019] Puppet (2019). 2019 State of DevOps Report. Technical report, Puppet.

[RADON, 2020] RADON (2020). RADON Continuous Testing Tool (CTT). https://github.com/radon-h2020/radon-ctt. Last visited on February 7, 2021.

[Radview, 2020] Radview (2020). WebLOAD - Smart Performance Testing. https://www.radview.com/. Last visited on February 7, 2021.

[Reichelt and Scheller, 2015] Reichelt, D. G. and Scheller, F. (2015). Improving Performance Analysis of Software System Versions Using Change-Based Test Selection. In *Proceedings of the 26th Symposium on Software Performance (SOSP)*, pages 1–3.

[Reitbauer et al., 2015] Reitbauer, A., Enzenhofer, K., Grabner, A., and Kopp, M. (2015). Agile Principles for Performance Evaluation - Java Enterprise Performance. https://www.dynatrace.com/resources/ebooks/javabook/adopting-agile-principles/. Last visited on February 7, 2021.

[RETIT GmbH, 2020] RETIT GmbH (2020). RETiT - Better performance for your software. https://www.retit.de/. Last visited on February 7, 2021.

[Rigor, 2014] Rigor (2014). Web Performance Monitoring - Rigor. http://rigor.com. Last visited on February 7, 2021.

[Röck et al., 2014] Röck, C., Harrer, S., and Wirtz, G. (2014). Performance Benchmarking of BPEL Engines: A Comparison Framework, Status Quo Evaluation and Challenges. In *Proceedings of the 26th Internation Conference on Software Engineering and Knowledge Engineering (SEKE)*, pages 1–4.

[Rodrigues et al., 2018] Rodrigues, I. P., Zorzo, A. F., Bernardino, M., and de Borba Campos, M. (2018). Usa-DSL: Usability Evaluation Framework for Domain-Specific Languages. In *Proceedings of the 31st ACM Symposium on Applied Computing (SAC)*, pages 2013–2021.

[Rosinosky et al., 2018] Rosinosky, G., Labba, C., Ferme, V., Youcef, S., Charoy, F., and Pautasso, C. (2018). Evaluating Multi-tenant Live Migrations Effects on Performance. *OTM Conferences*, 11229(8):61–77.

[Rudolph and Stitt, 2015] Rudolph, D. and Stitt, G. (2015). An interpolation-based approach to multi-parameter performance modeling for heterogeneous systems. In *Proceedings of the 26th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 174–180.

[Rufino et al., 2020] Rufino, V. Q., Nogueira, M., Avritzer, A., Menasché, D., Russo, B., Janes, A., Ferme, V., van Hoorn, A., Schulz, H., and Lima, C. (2020). Improving Predictability of User-Affecting Metrics to Support Anomaly Detection in Cloud Services. *IEEE Access*, 8(N/D):198152–198167.

[Schermann et al., 2018] Schermann, G., Cito, J., and Leitner, P. (2018). Continuous Experimentation - Challenges, Implementation Techniques, and Current Research. *IEEE Software*, 35(2):26–31.

[Schermann and Leitner, 2018] Schermann, G. and Leitner, P. (2018). Search-Based Scheduling of Experiments in Continuous Deployment. In *Proceedings of the 34th International Conference on Software Maintenance and Evolutio (ICSME)*, pages 485–495.

[Schermann et al., 2016] Schermann, G., Schöni, D., Leitner, P., and Gall, H. C. (2016). Bifrost - Supporting Continuous Deployment with Automated Enactment of Multi-Phase Live Testing Strategies. In *Proceedings of the 17th International Middleware Conference (Middleware)*, pages 12:1–12:14.

[Scheuner et al., 2014] Scheuner, J., Leitner, P., Cito, J., and Gall, H. C. (2014). Cloud Work Bench - Infrastructure-as-Code Based Cloud Benchmarking. In *Proceedings of the 6th IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 246–253.

[Schulz et al., 2019] Schulz, H., Okanović, D., van Hoorn, A., Ferme, V., and Pautasso, C. (2019). Behavior-Driven Load Testing Using Contextual Knowledge - Approach and Experiences. In *Proceedings of the 10th ACM/SPEC International Conference on Performance Engineering (ICPE)*, pages 265–272.

[Schulz and van Hoorn, 2020] Schulz, H. and van Hoorn, A. (2020). Representative Load Testing in Continuous Software Engineering: Automation and Maintenance Support. In Felderer, M., Hasselbring, W., Rabiser, R., and Jung, R., editors, *Proceedings of the Software Engineering*, pages 149–150.

[Schulz et al., 2020] Schulz, H., van Hoorn, A., and Wert, A. (2020). Reducing the maintenance effort for parameterization of representative load tests using annotations. *Software Testing, Verification and Reliability*, 30(1):e1712.

[Schwartz and Zaitsev, 2010] Schwartz, B. and Zaitsev, P. (2010). Goal-Driven Performance Optimization. https://www.percona.com/files/white-papers/goal-driven-performance-optimization.pdf. Last visited on February 7, 2021.

[Shatz, 2017] Shatz, I. (2017). Fast, Free, and Targeted: Reddit as a Source for Recruiting Participants Online. *Social Science Computer Review*, 35(4):537–549.

[Shoreditch Ops, 2017] Shoreditch Ops (2017). Artillery.io. https://artillery.io. Last visited on February 7, 2021.

[Silberschatz et al., 2013] Silberschatz, A., Galvin, P. B., and Gagne, G. (2013). *Operating System Concepts*. Wiley Publishing, 9th edition.

[Silva and Hines, 2016] Silva, M. and Hines, M. R. (2016). ibmcb/cbtool. https://github.com/ibmcb/cbtool. Last visited on February 7, 2021.

[Sim and Easterbrook, 2003] Sim, S. E. and Easterbrook, S. (2003). Using benchmarking to advance research: A challenge to software engineering. In *Proceedings of the 25th International Conference on Software Engineering (ICSE)*, pages 74–83.

[Simon, 2014] Simon, V. (2014). *Measurement and Analysis of Runtime-Metrics in a Continuous Integration Environment*. Bachelor's thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg.

[Skelton and Pais, 2019] Skelton, M. and Pais, M. (2019). *Team Topologies*. Organizing Business and Technology Teams for Fast Flow. IT Revolution.

[Skouradaki et al., 2015a] Skouradaki, M., Ferme, V., Leymann, F., Pautasso, C., and Roller, D. H. (2015a). "BPELanon": Protect business processes on the cloud. In *Proceedings of the 5th International Conference on Cloud Computing and Services Science (CLOSER)*, pages 241–250.

[Skouradaki et al., 2016] Skouradaki, M., Ferme, V., Pautasso, C., Leymann, F., and van Hoorn, A. (2016). Micro-Benchmarking BPMN 2.0 Workflow Management Systems with Workflow Patterns. In *Proceedings of the 14th Business Process Management Conference (BPM)*, pages 67–82.

[Skouradaki et al., 2014] Skouradaki, M., Roller, D., Leymann, F., Ferme, V., and Pautasso, C. (2014). Technical Open Challenges on Benchmarking Workflow Management Systems. In *Proceedings of the 25th Symposium on Software Performance (SOSP)*, pages 105–112.

[Skouradaki et al., 2015b] Skouradaki, M., Roller, D. H., Leymann, F., Ferme, V., and Pautasso, C. (2015b). On the Road to Benchmarking BPMN 2.0 Workflow Engines. In *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering (ICPE)*, pages 301–304.

[SmartBear, 2016] SmartBear (2016). Load Complete. https://smartbear.com/product/loadcomplete/overview/. Last visited on February 7, 2021.

[SmartBear, 2020] SmartBear (2020). The State of API 2020 Report. https://smartbear.com/resources/ebooks/the-state-of-api-2020-report//. Last visited on February 7, 2021.

[Smith and Williams, 2001] Smith, C. U. and Williams, L. G. (2001). *Performance Solutions*. A Practical Guide to Creating Responsive, Scalable Software. Addison-Wesley Professional.

[Sobel et al., 2008] Sobel, W., Subramanyam, S., Sucharitakul, A., and Nguyen, J. (2008). Cloudstone: Multi-platform, multi-language benchmark and measurement tools for web 2.0. In *Proceedings of the 17th ACM International Conference on Cloud Computing and Applications*, pages 1–10.

[Sopeco, 2014] Sopeco (2014). sopeco/DynamicSpotter. https://github.com/sopeco/DynamicSpotter. Last visited on February 7, 2021.

[Spafford and Vetter, 2012] Spafford, K. and Vetter, J. S. (2012). Aspen - a domain specific language for performance modeling. In *Proceedings of the 24th ACM International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–10.

[SPEC, 2016] SPEC (2016). SPEC RG DevOps Performance Working Group. https://research.spec.org/working-groups/rg-devops-performance.html. Last visited on February 7, 2021.

[Specification, 1999] Specification, W. M. C. (1999). *Workflow Management Coalition, Terminology and Glossary (Document No. WFMC-TC-1011)*. Workflow Management Coalition Specification.

[Spinellis, 2001] Spinellis, D. (2001). Notable design patterns for domain-specific languages. *Journal of Systems and Software*, 56(1):91–99.

[Spinellis, 2016] Spinellis, D. (2016). Being a DevOps Developer. *IEEE Software*, 33(3):4–5.

[Spinner et al., 2015] Spinner, S., Casale, G., Brosig, F., and Kounev, S. (2015). Evaluating approaches to resource demand estimation. *Performance Evaluation*, 92(C):51–71.

[StackRox, 2020] StackRox (2020). Kubernetes and Container Security and Adoption Trends. Technical report, StackRox.

[Streitz et al., 2018] Streitz, A., Barnert, M., Kienegger, H., and Krcmar, H. (2018). Performance Improvement Barriers for SAP Enterprise Applications - An Analysis of Expert Interviews. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering (ICPE)*, pages 223–228. Technical University of Munich, Munich, Germany.

[Tolledo, 2014] Tolledo, R. (2014). Gatling: Take Your Performance Tests to the next Level. https://www.thoughtworks.com/insights/blog/gatling-take-your-performance-tests-next-level. Last visited on February 7, 2021.

[Tricentis, 2020] Tricentis (2020). State of Open Source Testing. https://www.tricentis.com/state-of-open-source-2020/. Last visited on February 7, 2021.

[Trubiani et al., 2018] Trubiani, C., Jamshidi, P., Cito, J., Shang, W., Jiang, Z. M., and Borg, M. (2018). Performance Issues? Hey DevOps, Mind the Uncertainty. *IEEE Software*, 36(2):110–117.

[Tsakiltsidis et al., 2016] Tsakiltsidis, S., Miranskyy, A., and Mazzawi, E. (2016). On Automatic Detection of Performance Bugs. In *Proceedings of the 28th IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 132–139.

[van Deursen et al., 2000] van Deursen, A., Klint, P., and Visser, J. (2000). Domain-specific languages. *ACM SIGPLAN Notices*, 35(6):26–36.

[Van Gelder et al., 2014] Van Gelder, L., Das, P., Janssen, H., and Roels, S. (2014). Comparative study of metamodelling techniques in building energy simulation: Guidelines for practitioners. *Simulation Modelling Practice and Theory*, 49(1):245–257.

[van Hoorn et al., 2012] van Hoorn, A., Waller, J., and Hasselbring, W. (2012). Kieker - a framework for application performance monitoring and dynamic software analysis. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering (ICPE)*, pages 247–248.

[Veeraraghavan et al., 2016] Veeraraghavan, K., Meza, J., Chou, D., Kim, W., Margulis, S., Michelson, S., Nishtala, R., Obenshain, D., Perelman, D., and Song, Y. J. (2016). Kraken - Leveraging Live Traffic Tests to Identify and Resolve Resource Utilization Bottlenecks in Large Scale Web Services. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 635–651.

[Voelter et al., 2013] Voelter, M., Benz, S., Dietrich, C., and Engelmann, B. (2013). *DSL engineering: Designing, implementing and using domain-specific languages*. CreateSpace Independent Publishing Platform.

[Vögele et al., 2014] Vögele, C., Brunnert, A., Danciu, A., Tertilt, D., and Krcmar, H. (2014). Using Performance Models to Support Load Testing in a Large SOA Environment. In *Proceedings of the 3rd International Workshop on Large Scale Testing (LT)*, pages 5–6.

[Waller et al., 2015] Waller, J., Ehmke, N. C., and Hasselbring, W. (2015). Including Performance Benchmarks into Continuous Integration to Enable DevOps. *ACM SIGSOFT Software Engineering Notes*, 40(2):1–4.

[Walter, 2018] Walter, J. (2018). *Automation in Software Performance Engineering Based on a Declarative Specification of Concerns*. PhD thesis, University of Würzburg.

[Walter et al., 2016] Walter, J., van Hoorn, A., Koziolek, H., Okanović, D., and Kounev, S. (2016). Asking "What"?, Automating the "How"? - The Vision of Declarative Performance Engineering. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering (ICPE)*, pages 91–94.

[Wang, 2006] Wang, Y. (2006). *Automating experimentation with distributed systems using generative techniques*. PhD thesis, University of Colorado at Boulder.

[Weaveworks, 2018] Weaveworks (2018). Flagger. https://flagger.app/. Last visited on February 7, 2021.

[Weber et al., 2015] Weber, I., Bass, L., and Zhu, L. (2015). DevOps A Software Architects Perspective. Addison-Wesley Professional.

[Westermann, 2014] Westermann, D. (2014). *Deriving Goal-oriented Performance Models by Systematic Experimentation*. PhD thesis, Karlsruhe Institute of Technology.

[Wettinger et al., 2016] Wettinger, J., Breitenbücher, U., Falkenthal, M., and Leymann, F. (2016). Collaborative gathering and continuous delivery of DevOps solutions through repositories. *Computer Science - Research and Development*, 31(4):1–10.

[Willnecker et al., 2015] Willnecker, F., Brunnert, A., Gottesheim, W., and Krcmar, H. (2015). Using Dynatrace Monitoring Data for Generating Performance Models of Java EE Applications. In *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering (ICPE)*, pages 103–104.

[Willnecker et al., 2016] Willnecker, F., Kroß, J., and van Hoorn, A. (2016). Performance and the Pipeline. http://blog.ieeesoftware.org/2016/11/performance-and-pipeline.html. Last visited on February 7, 2021.

[Woodside et al., 2013] Woodside, M., Petriu, D. C., Merseguer, J., Petriu, D. B., and Alhaj, M. (2013). Transformation challenges: from software models to performance models. *Software & Systems Modeling*, 13(4):1529–1552.

[Wynne and Hellesoy, 2012] Wynne, M. and Hellesoy, A. (2012). *The Cucumber Book: Behaviour-Driven Development for Testers and Developers*. Pragmatic Bookshelf.

[Yan et al., 2012] Yan, M., Sun, H., Wang, X., and Liu, X. (2012). WS-TaaS - A Testing as a Service Platform for Web Service Load Testing. In *Proceedings of the 18th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 456–463.

[Yang et al., 2005] Yang, L. T., Ma, X., and Mueller, F. (2005). Cross-Platform Performance Prediction of Parallel Applications Using Partial Execution. In *Proceedings of the 17th ACM International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–11.

[Yu et al., 2020] Yu, L., Alégroth, E., Chatzipetrou, P., and Gorschek, T. (2020). Utilising CI environment for efficient and effective testing of NFRs. *Information and Software Technology*, 117(1):106–199.

[Zalavadia, 2016] Zalavadia, S. (2016). Integrating Quality Assurance Throughout the Deployment Pipeline. https://dzone.com/articles/integrating-qa-throughout-the-deployment-pipeline. Last visited on February 7, 2021.

[Zheng et al., 2015] Zheng, X., Ravikumar, P., John, L. K., and Gerstlauer, A. (2015). Learning-based analytical cross-platform performance prediction. In *Proceedings of the 15th IEEE International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pages 52–59.

[Zhu et al., 2016] Zhu, L., Bass, L., and Champlin-Scharff, G. (2016). DevOps and its practices. *IEEE Software*, 33(3):32–34.

[Zorn, 2018] Zorn, C. (2018). *Concern-driven Reporting of Declarative Performance Analysis Results Using Natural Language and Visualization*. Bachelor's thesis, University of Stuttgart.

Acronyms

RESTful RESTful. viii, 6, 8, 33, 41, 49, 57, 61, 63, 109, 113, 143, 147, 189, 226, 272, 273, 292, 293, 306, 322, 323, 333, 366, 367, 369, 372

AI Artificial Intelligence. 53

API Application Programming Interface. xvi, 11, 28, 41, 49, 51, 53, 113, 226, 229, 232, 234, 238, 267, 268, 273, 276–281, 283, 291–293, 348, 351, 353, 357, 365, 366, 369, 373

APM Application Performance Management. 39, 53, 369, 375

AST Abstract Syntax Tree. 216

BDD Behavior-driven Development. 366

BDLT Behavior-driven Load Testing. xxiii, 353, 354

BPEL Business Process Execution Language. 43

BPMN 2.0 Business Process Model and Notation 2.0. viii, 6, 8, 12, 21, 33, 42, 43, 49, 57, 61, 63, 109, 113, 139, 143, 147, 149, 191, 215, 225, 272, 285, 287, 333–335, 337, 338, 347, 349, 350, 367, 372

CD Continuous Delivery. 38, 39, 48

CDS Continuous Delivery System. 104, 154, 158, 213

CI Continuous Integration. 38, 39, 48, 365, 366

CICD Continuous Integration and Delivery. 48, 51, 53, 104, 106, 107, 292, 293, 312, 316, 322, 323, 367

CIS Continuous Integration System. 104, 154, 158, 213

CLI Command-line Interface. 8, 11, 51, 53, 229, 231, 232, 238, 272, 291–293, 352, 363, 365, 366, 369

CPU Central Processing Unit. 51, 69, 75, 82, 86, 115, 116, 120, 134, 136, 139, 177, 178, 215, 266, 267, 278, 279, 282, 284, 286, 287, 289

CSDL Continuous Software Development Lifecycle. vii, viii, xiii–xvi, xx, xxi, xxiii–xxv, 3, 6, 8–11, 13, 14, 24, 27, 31, 33, 36–41, 44, 45, 47–51, 53–57, 62, 63, 103–107, 110, 111, 121, 131, 154–222, 229, 231, 257, 273, 292, 293, 298, 301–303, 306, 314, 315, 318, 322, 323, 337, 339, 340, 342–345, 347, 352, 363, 365, 367–371, 373–376

CSV Comma-separated values. 51, 145

DBMS Database Management System. 43, 95, 102, 103, 192, 224, 238, 266, 282, 285, 348

DevOps Development and Operations. vii, xiii, xiv, xvi, 3, 4, 6–8, 10, 13, 16, 30, 31, 33, 36–41, 45, 48, 49, 57, 61, 103–111, 113, 114, 273, 292, 305, 314, 316, 321–324, 365–367

DISK Disk Storage. 75, 82, 86, 90, 102, 139, 215, 267, 286, 287

DPE Declarative Performance Engineering. vii, xiv, xxiii, 4–9, 12, 13, 15, 33, 34, 36, 41, 43–45, 49, 54, 57, 61, 62, 69–71, 73, 114–116, 159, 223, 301, 306, 323, 332, 338, 340, 344, 350, 353, 359, 363, 366, 367, 369–371, 373, 374, 376

DSL Domain Specific Language. vii, viii, xv–xviii, xx, xxi, xxiii–xxv, 7–14, 24, 35, 36, 44, 46–48, 57, 112–223, 225, 226, 232, 233, 236–238, 242, 245, 246, 248, 251, 265, 272–274, 284, 285, 293, 297–332, 336–342, 345, 347, 349–354, 357, 359, 363–370, 372, 373, 375, 376, 381, 389, 394, 395, 399

GUI Graphical User Interface. 50–52, 54, 100, 101, 271

HTML Hypertext Markup Language. 51

HTTP Hypertext Transfer Protocol. 50, 125, 277

ID Identifier. 277

IDE Integrated Development Environment. 36, 214, 233, 332, 373, 375

IO Input/Output. 267, 286

IP Internet Protocol. 149, 232, 271, 282, 283

IT Information Technology. vii, 4, 36, 37, 46, 297

JDBC Java Database Connectivity. 50

JMS Java Message Service. 50

JSON JavaScript Object Notation. 146, 266, 277, 279

JVM Java Virtual Machine. 64, 82, 111

KPI Key Performance Indicator. 225, 227, 229, 230, 238, 245, 350

MAE Mean Absolute Error. 141

MARS Multivariate Adaptive Regression Spline. 138

NET Network. 75, 82, 86, 89, 95, 139, 215, 267, 286, 287

OS Operating System. 82, 136

R.G. Research Goal. xxiii, 13, 14, 345

R.G. 1 Research Goal 1. xiii, 6, 9, 13, 14, 61, 111

R.G. 2 Research Goal 2. xiii, 7, 9, 10, 13, 14, 113, 222, 314, 338, 345

R.G. 3 Research Goal 3. xiii, 7, 10, 13, 14, 293, 314, 332, 338, 345

R.G. 4 Research Goal 4. xiii, 8, 10, 11, 13, 14, 111, 222, 293, 339, 345

RAM Random Access Memory. 75, 82, 86, 90, 115, 116, 120, 134, 136, 137, 139, 177, 178, 215, 267, 286, 287

REST Representational State Transfer. 11, 113, 276, 279, 348

RQ Research Question. 314, 317, 332

SLA Service Level Agreement. 4, 41, 53, 63, 66, 81, 82, 92, 105, 343

SLO Service Level Objectives. 85, 91, 92, 105, 107

SOAP Simple Object Access Protocol. 50

SQL Structured Query Language. 267

SUS System Usability Scale. 318

SUT System Under Test. xiv, xv, xxi, xxiv–xxvi, 4, 8–11, 29, 30, 35, 47, 48, 51–54, 61–63, 67, 68, 70–103, 105–111, 115–122, 125, 126, 130, 131, 133, 134, 136–141, 143, 146–149, 153, 156, 158, 160, 170, 180, 181, 183, 187, 188, 192, 197–199, 206–208, 210–217, 222–227, 229, 230, 232–234, 236, 242, 244, 246, 248, 250, 251, 257, 259–261, 263, 265, 267–269, 271, 272, 274, 276, 278, 279, 281, 282, 284–286, 289, 291, 292, 302, 307, 309, 313, 324, 326, 336, 338–343, 350, 351, 353, 356, 357, 364, 368, 369, 375, 376, 423, 465

UML Unified Modeling Language. 46

URI Uniform Resource Identifier. 276

URL Uniform Resource Locator. 145, 146, 277, 300, 316

VCS Version Control System. 103, 104, 114, 156, 160

VM Virtual Machine. 47, 54

WfMS Workflow Management System. xxiii, xxv, 8, 12, 20, 21, 24, 33, 42, 43, 49, 54, 57, 61, 63, 93, 109, 113, 125, 139, 143, 147, 149, 191, 192, 225, 268, 272, 285, 287, 333–335, 337, 338, 347–350, 367, 372

XML Extensible Markup Language. 47, 272

YAML YAML Aint Markup Language. xv, xviii, xxiv–xxvi, 46, 47, 51, 159–161, 163–166, 168–171, 173–184, 187–189, 191, 193–195, 197, 199, 201, 203, 206, 208, 210–213, 216, 217, 222, 232, 280, 281, 301–303, 306–308, 312, 323, 332, 381, 389, 394, 395, 399, 408, 409, 411, 412, 414, 416, 418, 420, 421, 423, 424, 426, 430, 431, 433, 435, 437, 450, 451, 453, 454, 456, 458, 460, 462, 463, 465, 466, 468, 472, 473, 475