A Declarative Approach for Performance Tests Execution in Continuous Software Development Environments

Vincenzo Ferme, Cesare Pautasso
Software Institute, Faculty of Informatics, USI Lugano, Switzerland

ABSTRACT

Software performance testing is an important activity to ensure quality in continuous software development environments. Current performance testing approaches are mostly based on scripting languages and framework where users implement, in a procedural way, the performance tests they want to issue to the system under test. However, existing solutions lack support for explicitly declaring the performance test goals and intents. Thus, while it is possible to express how to execute a performance test, its purpose and applicability context remain implicitly described. In this work, we propose a declarative domain specific language (DSL) for software performance testing and a model-driven framework that can be programmed using the mentioned language and drive the end-to-end process of executing performance tests. Users of the DSL and the framework can specify their performance intents by relying on a powerful goal-oriented language, where standard (e.g., load tests) and more advanced (e.g., stability boundary detection, and configuration tests) performance tests can be specified starting from templates. The DSL and the framework have been designed to be integrated into a continuous software development process and validated through extensive use cases that illustrate the expressiveness of the goal-oriented language, and the powerful control it enables on the end-to-end performance test execution to determine how to reach the declared intent.

ACM Reference Format:
Vincenzo Ferme, Cesare Pautasso. 2018. A Declarative Approach for Performance Tests Execution in Continuous Software Development Environments. In *ICPE '18: ACM/SPEC International Conference on Performance Engineering, April 9–13, 2018, Berlin, Germany.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3184407.3184417

1 INTRODUCTION AND MOTIVATION

The Software is pervasively assembled across different businesses, by professional figures with different skills [22] and with often-changing technologies [9]. Agile software development is widely used nowadays as a process to conceive, design, develop and operate complex software systems. Agile practices aim at reaching software deployment to production fast and often, to collect feedback from the users to be used in the next release of the same software, enabling continuous software development (CSD) [23].

To guarantee the quality of the software released to production, software development pipelines automation, and pervasive automated testing are vital for succeeding in releasing reliable software [12], so that people involved in the process can get continuous feedback. Performance testing is a kind of testing activity performed within development pipelines and by its complex nature, requires expertise to define performance tests, configure and manage the load infrastructures, the automated deployment of the software in different configurations, and the performance data collection and analysis. Many tools have been proposed to help the professional figures involved in the CSD process to implement performance testing, as for example to specify and execute performance tests (e.g., JMeter[1]), managing the load infrastructure (e.g., Faban[2]), to automate the deployment of the system under test (SUT) (e.g., Docker[3]), and comprehensive solutions to help users in the entire end-to-end performance test execution and results' analysis (e.g., DataMill [14]).

1.1 Context and Motivation

In this paper we focus in particular on CSD lifecycles, where the developed software is represented by (Micro)services, and we look at automating the execution of performance tests issued against the APIs, particularly REST APIs, exposed to the users. The main characteristics of this context are: (1) professional figures having diversified roles and heterogeneous performance knowledge [22], with control and responsibility on part of the developed services throughout their entire lifecycle from development to production; (2) parallel development of different services realizing the developed software, relying on one or more project repositories, and different branches [23] to version code and related artifacts; (3) continuous evolution of the developed software, by leveraging users' feedback and production data about application behavior; (4) automation of release pipelines, including code quality checks, build, test, packaging, delivery and in some cases deployment as well [23].

As argued by us [8], and by other researchers (e.g., [4]), execution of performance tests should be automated, flexible,

ICPE '18, April 9–13, 2018, Berlin, Germany

ACM ISBN 978-1-4503-5095-2/18/04…$15.00
https://doi.org/10.1145/3184407.3184417

[1] http://jmeter.apache.org, last visited April 11, 2018
[2] http://faban.org, last visited April 11, 2018
[3] https://www.docker.com, last visited April 11, 2018

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. Usually in CSD, users define performance tests that 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.

While for most of the available tools and solutions, users define performance tests using scripts or code, we argue that in this context a declarative model-driven approach [8], exposed to the users by means of a domain specific language (DSL), could help to control the end-to-end process of executing performance tests, by making the purpose and applicability contexts of defined performance tests explicit. Empowered by the DSL, different professional figures can express their performance intents in terms of performance goals, for example, defined by more expert performance people. They can then rely on an extensible framework that can automate and control the end-to-end execution of tests, by following the directives specified in the model behind the DSL definition.

1.2 Requirements

Given the described context, we identified the following requirements the proposed approach embraces:

(1) *declarative* specification of performance tests, performance goals and SUT deployment and configuration;
(2) *automated and model-driven execution* of the end-to-end performance test lifecycle;
(3) *extensibility* of the DSL and the automation infrastructure, to cope with evolving needs in CSD, and custom requirements of different contexts and applications.

We propose two main contributions to enhance the state of the art in this context. The first contribution is a declarative DSL for performance testing (Sect. 3), allowing the users to declare the goal of the specified performance test, as well as control the deployment configuration of the SUT. The second one is a model-driven framework (Sect. 4) that can be programmed using the mentioned DSL, and automate the end-to-end execution of performance tests, by executing all the activities that are needed to answer to the performance intent of the user, declared as a goal of the performance test.

A declarative approach to performance tests execution, implemented in a DSL and a framework, to specify the intent of performance tests, and to describe the SUT deployment and configuration enables the production of shared artifacts that can be exchanged among development team members with different expertise and profiles. We opted for a declarative DSL, so that users can rely on a domain model closer to the performance testing terminology, and the code that defines how to execute the test is actually built into the framework that represents the runtime of the proposed DSL. We do not want the users to necessarily care about how the actual performance test execution is implemented, but we want them to be able to define performance activities and control 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. By abstracting, the expressiveness of the language compared to imperative code is reduced, because only specific concepts are integrated into the DSL. For this reason we made the DSL, and the model-driven framework actually driving the execution of performance tests, open and extensible to new use cases and different needs.

The rest of the paper is structured as follows: in Sect. 2 we present related work on declarative performance testing and DSLs, in Sect. 3 we present the proposed declarative DSL, in Sect. 4 the model-driven framework based on the same and in Sect. 5 use cases in defining and executing performance tests based on the presented DSL and the framework, in Sect. 6 we conclude the paper and briefly present planned future work.

2 RELATED WORK

Declarative Software Performance Testing - Declarative software performance engineering, part of which is also related to performance testing, has been presented as part of the DECLARE project[4] by Walter et al. [20]. The DECLARE project "envisions to reduce the current abstraction gap between the level on which performance-relevant concerns are formulated 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 modelled by other systems, while the focus of our work is in applying declarative approaches for performance test specification and automated execution. In this context, Westermann [21] present the concept of goal-driven performance testing, mainly related to smart exploration of the performance spaces for different configurations of software systems (i.e., the space described by all the possible combinations of the configuration variables). This is realized with a DSL, and a runtime framework for automatic performance test execution. The main difference compared to our approach is the context of application, and the focus of the declarative goal-driven definitions, than in our case are related to performance testing of container-packaged (Micro)services

[4] http://www.dfg-spp1593.de/declare, last visited April 11, 2018

within CSD lifecycles, and open to answer different performance intents users might have. Other relevant work proposing both a DSL and a framework, are Cloud Work Bench [17], Crawler [5], CloudPerf [13] and Jagger[5], tools for benchmarking the performance of the services offered by cloud providers. The first three tools propose a declarative DSL tailored to Cloud benchmarking, then targeting a different domain than the one proposed in the context of this work. Although Jagger's DSL allows users to declaratively specify success criteria of tests, in our goal-oriented approach we additionally support describing the intent of performance tests.

Other related work propose specific approaches to answer different kinds of performance intents, as for example capacity planning [18], and performance optimization in the presence of constraints [6]. 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.

Performance Testing Tools in CSD - Experimentation and continuous automated checking of performance quality criteria, are important activities in CSD environments. Performance experimentation, and experimentation in general, is discussed in different research and industry work as an important activity in continuous development, e.g., by Google with Vizier [11], AutoPerf [1], DataMill [14] and the approaches by Omar et al. [15], Westermann [21] and Cloud Work Bench [17]. They propose languages and framework to help users simplifying exploratory test definition, automating performance test execution, and ensuring rigorous performance data analysis. The solution we propose builds on existing tools, and integrates them to achieve full control over the entire performance testing lifecycle with a declarative DSL.

Different solutions have been proposed for continuous automated checking of performance quality criteria. Blazemeter[6] integrates standard performance testing tools in CSD, by providing the load infrastructure, and a software as a service platform on which automatically schedule and execute performance tests. Other tools rely on performance management platforms[7] to collect performance metrics and validate them over time, others apply live testing [10, 16] for incremental roll-out of new versions of (Micro)service applications according to their performance behaviour. Others continuously check for regressions of different software performance metrics [3, 19] after every set of relevant commits. Overall the plethora of solutions is rich and diversified, and in this work we discuss how to integrate them at a higher level of abstraction with a declarative DSL allowing the user to specify their performance intent, and then automate them in a framework for automating the goal-driven end-to-end process of performance test execution.

3 DECLARATIVE PERFORMANCE DSL

The DSL is used to specify the intent of performance tests and control their entire end-to-end execution process in a declarative manner, and it is particularly tailored to container-packaged (Micro)service systems. In general 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, test bed management and analysis of performance data [2, 21]. Our DSL provides the mentioned features, and adds a goal-oriented, and declarative specification of performance intents as well. The declarative nature of the DSL, enables the users to start from provided templates for defining performance tests. They can then update the tests according to changing requirements, or dispose them in favour of new specifications. This is possible without the need of rebuilding the test or changing code, but by manually or programmatically updating its specification.

3.1 Meta-Model: Overview

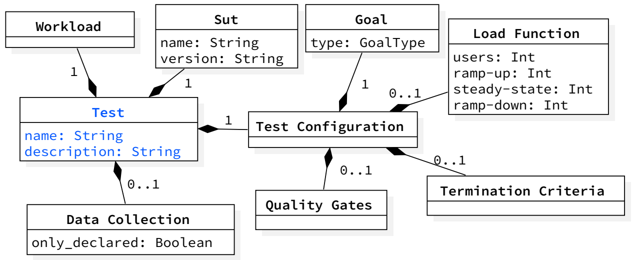

Figure 1: **DSL: The Test Meta-Model**

The main entity in the language is *test*, which as shown in Fig. 1 groups together the *workload*, the *sut*, the *data collection* and the test *configuration*. As part of the configuration, users can indicate the *goal*, the corresponding *load function* as well as the criteria to drive the automated execution of the test needed to achieve the goal: *termination criteria* to control conditions to declare the test completed, and *quality gates* to determine if wanted quality criteria have been achieved or not by the SUT.

Additionally, the user can specify other core performance concepts, such as: 1) the *workload*, in terms of named sets of operations as well as parameters about the way to mix those sets of operations together with the *inter operations timing* specification; 2) details on the *SUT* such as the target endpoint of the test, deployment time configuration settings and specifications about the machines where to deploy the different services realizing the SUT. The actual deployment descriptor of the SUT can be specified using the Docker Cloud and Compose standards[8]; 3) *data collection services* to enable, so that client-side and server-side performance data can be collected.

[5] https://jagger.griddynamics.net, last visited April 11, 2018
[6] Blazemeter - https://www.blazemeter.com, and Taurus - http://gettaurus.org, last visited April 11, 2018
[7] http://rigor.com, last visited April 11, 2018

[8] Docker Cloud - https://docs.docker.com/docker-cloud/apps/stack-yaml-reference/ and Docker Compose - https://docs.docker.com/compose/compose-file/, last visited April 11, 2018

```
name: The test name
description: The test description # Optional
configuration:
   goal:
   load_function:  # Optional IF specified in goal
   termination_criteria:  # Optional
   quality_gates:  # Optional
sut:
workload:
data_collection:  # Optional
```

Listing 1: **DSL: The Test YAML Format Overview**

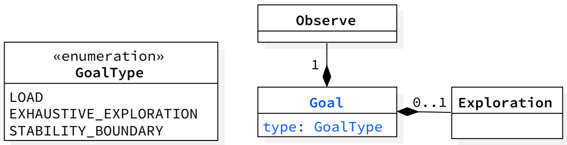

Figure 2: **DSL: The Goal Meta-Model**

In Listing 1 we present the actual format, omitting some details, of the DSL the user is writing to specify a test using the YAML[9] syntax, that is both human and machine readable.

3.2 Meta-Model: Main Entities

Goal - The *goal* is part of the test *configuration* and is used to declaratively specify the users' performance intent by relying on given performance goals (Fig. 2), such as executing a load test or exploring the performance or the stability of the application in a given configuration space.

We defined a taxonomy of goals [8], where we distinguish goals by their different levels of abstractions: meta goals (e.g., comparing the performance of different systems using a benchmark), goals (e.g., capacity planning, stability boundary testing) and base goals (e.g., load test, configuration test). The current main focus of the DSL is to support standard performance tests, such as load test, and exploratory performance testing, thus the goal types currently provided by the language are the `load`, and `exhaustive_exploration` (similar to configuration test) base goals, and the `stability_boundary` goal. We decided to support exploratory testing, because we, and other researchers [10, 11], argue that in CSD, with continuous evolution and feedback, it is very important to be able to explore performance of different system's configuration or alternative solutions at any moment in the development lifecycle, to gain insights on the behaviour of the application.

Depending on the selected goal it might be necessary to specify also other information to automate the test execution, e.g., adding an exploration strategy. The `load` goal is a standard load test and does not require additional configuration. The `exhaustive_exploration` and the `stability_boundary` goals execute multiple experiments by exploring the *exploration_space* according to some *exploration_strategy* (e.g., randomly or with a binary search). For the `stability_boundary` it is also required to specify the `stability_criteria`, where the user can define stability conditions using the same semantics of quality gates, presented later in this section.

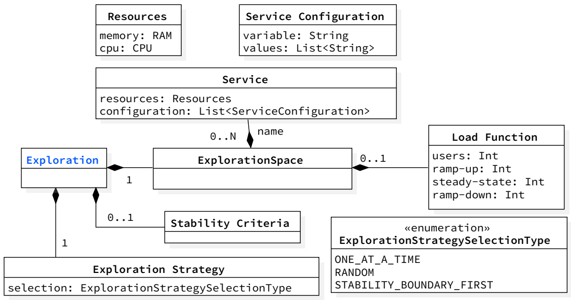

Figure 3: **DSL: The Exploration Meta-Model**

Exploration Space - The *exploration_space* defines the variables that can be changed between experiments and their possible values (Fig. 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. The user can directly specify the values to set over different experiments, or specify ranges to navigate with given step functions, that can, for example, apply addition, subtraction, multiplier, division and power for a numeric variable.

We support changes in the load function by number of users, and the ramp-up, steady-state and ramp-down. This is useful when the goal is to explore how the system behaves under different loads. The configuration through environment variables consists of any *variable-values* pair specified by the user. For resources we currently support setting the CPU and RAM allocation, however, these can be extended to other resources supported by the Docker Cloud and Compose standards, e.g., `i/o` speed as well as other container orchestration and management frameworks (e.g., Kubernetes[10]). The user decides how to traverse the *exploration_space*, by selecting an *exploration_strategy*. Each strategy determines the order in which different experiments are executed to achieve the goal of the test. Currently supported are the `one-at-a-time` strategy, that select the experiment one after the other following the different dimensions of the *exploration_space*, `random` strategy that schedules the experiments in random order, and `stability-boundary-first` strategy that uses a binary search to trace the stability boundary. 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 [21].

Termination Criteria - The exploration, and in general a performance test execution might incur into failures, and last for a long amount of time. Thus the *termination_criteria* are

[9] http://www.yaml.org, last visited April 11, 2018

[10] https://kubernetes.io, last visited April 11, 2018

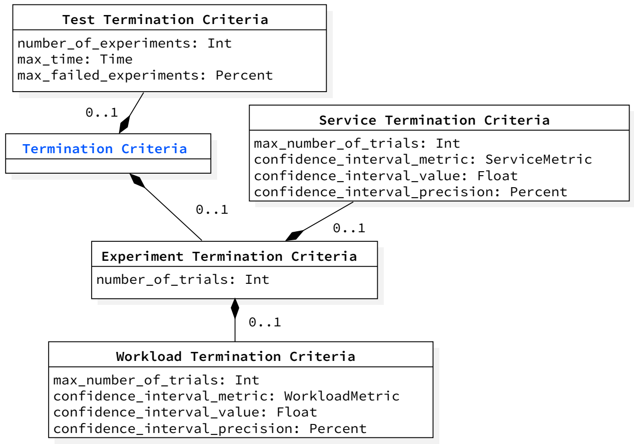
Test Termination Criteria
number_of_experiments: Int
max_time: Time
max_failed_experiments: Percent
0..1
Termination Criteria
Service Termination Criteria
max_number_of_trials: Int
confidence_interval_metric: ServiceMetric
confidence_interval_value: Float
confidence_interval_precision: Percent
0..1
0..1
Experiment Termination Criteria
number_of_trials: Int
0..1
Workload Termination Criteria
max_number_of_trials: Int
confidence_interval_metric: WorkloadMetric
confidence_interval_value: Float
confidence_interval_precision: Percent

Figure 4: **DSL: The Termination Criteria Meta-Model**

used to determine when the test execution process can be considered to be completed (Fig. 4). The termination criteria apply to different entities, namely: the entire test and the navigation of the exploration space, and the different trials of the experiments, i.e., repeated experiment executions used to collect more precise measurements and to cope with the intrinsic variability of performance, that get executed as part of the test.

The currently supported test termination criteria are: a fixed limit on the number of experiments to be executed, the maximum amount of test running time (`max_time`), and the maximum allowed number of experiments marked as failed (using a percentage of the overall number of experiments to be executed) without providing any result (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*.

We support two different experiment termination criteria, which control the number of trials. Thus we support the possibility to statically specify a fixed upper bound on the number of trials to be executed for each experiment, or a target confidence interval (c.i.) to be achieved for one metric of interest on a workload or a service at a given precision, and thus dynamically determine the number of trials to be executed to reach the given c.i. up to the given maximum number of trials. In general, if multiple termination criteria are specified, they are all applied and the test/experiment terminates as soon as one criteria is satisfied.

The final state of the test/experiment depends on whether the termination criteria corresponds to describing a condition under which the goal has been reached or a condition that represents the impossibility of reaching the goal. For example, if the goal is to perform a given number of experiments or trials, reaching the *number_of_experiments*, *number_of_trials* will result in a successful test or experiment. However, if it was not possible to reach the required confidence interval and instead the upper limit on the number of trials was reached, this results in the failure of the experiment. Likewise, if the duration exceeds the *max_time* or the number of failures reaches the *max_failed_experiment* limit, the corresponding test will be marked as failed.

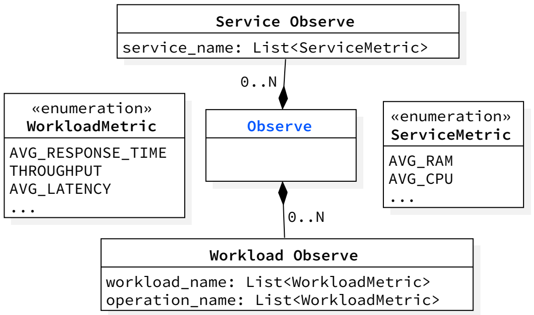

Figure 5: **DSL: The Observe Meta-Model**

Observe - The performance metrics of interest for the test are enumerated within the *observe* entity (Fig. 5). 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 on its operations. Some available metrics are: response time, latency, throughput for each single operation and for the entire workload, for each trial and aggregated at experiment level, as well as time series over the entire load function time. Server-side metrics can be observed on specific services realizing the SUT, and some available metrics are related to: RAM, CPU, IO, network utilization. SUT specific metrics can be defined as well, and integrated in the framework. Other metrics can be computed on top of logs collected from the SUT. On all the metrics we also make available descriptive statistics, and statistical tests to check for the homogeneity of the collected data (e.g., coefficient of variation[11], and Levene's test[12]), that is for example useful to validate whether the collected data over multiple trials of the same experiment exposes the same behaviour. The user can observe specific metric (e.g., throughput) or statistics (e.g., average CPU utilization), or the entire set of metric and statistics computed on an entity (e.g., all the metrics and descriptive statistics of CPU).

Quality Gates - Quality gates help with integrating the tests in CSD, by enabling the possibility to express performance requirements for the SUT, for the current defined test. They declare which are the successful and failure conditions of a test so that these can be checked automatically (Fig. 6).

They currently include success conditions (i.e., conditions that lead to mark a test as successful) on: all the observable metrics, and relative aggregated statistics, on any of the services realizing the SUT, and on the workload issued to the system, to validate that the issued workload satisfied the specified requirements. The conditions (one of >, <, >=, <=, =) that can be specified on the metrics allow comparing the value of a metric or a statistic with a static value,

[11] http://www.ats.ucla.edu/stat/mult_pkg/faq/general/coefficient_of_variation.htm, last visited April 11, 2018
[12] http://www.itl.nist.gov/div898/handbook/eda/section3/eda35a.htm, last visited April 11, 2018

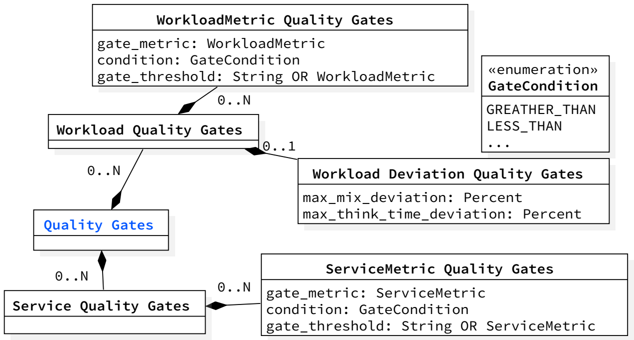
WorkloadMetric Quality Gates
gate_metric: WorkloadMetric
condition: GateCondition
gate_threshold: String OR WorkloadMetric
0..N
«enumeration»
GateCondition
GREATHER_THAN
LESS_THAN
...
Workload Quality Gates
0..1
Workload Deviation Quality Gates
max_mix_deviation: Percent
max_think_time_deviation: Percent
0..N
Quality Gates
0..N
Service Quality Gates
0..N
ServiceMetric Quality Gates
gate_metric: ServiceMetric
condition: GateCondition
gate_threshold: String OR ServiceMetric

Figure 6: **DSL: The Quality Gates Meta-Model**

or with another metric or statistic. If more than one condition is specified, they are all evaluated and in order for a test to be considered successful, all the conditions have to be satisfied.

The conditions that can be specified on the issued workload, also enable the user to specify the maximum allowed deviation from the defined mix and the maximum allowed deviation from the specified think time for the simulated users. The YAML format depends on the actual mix that is selected by the user. 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 think time.

Currently all the quality gates are verified after the execution of an experiment, thus are applied to each experiment, and consequently to each test if the gates are defined on a test level metric.

Quality gates complement termination criteria. Quality gates are evaluated after successful executions of experiments to determine whether the test succeeded. Termination criteria instead control and act on the test execution process and determine the final state of execution of experiments as a function of the outcome of the corresponding trials. They are also used to limit the execution time of tests with bounds on the maximum runtime or maximum number of failures.

Workload - The workload entity (Fig. 7) 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. 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 catalogue and buying goods, and a set of operations named “admins” of website admins adding new items to the catalogue. The actual format used to describe them in the DSL is omitted for space reason.

For each set of operations it is possible to specify its popularity, representing the percentage of requests that should be issued to the SUT from the given named workload. Within a single set of operations, it is possible to indicate how to mix

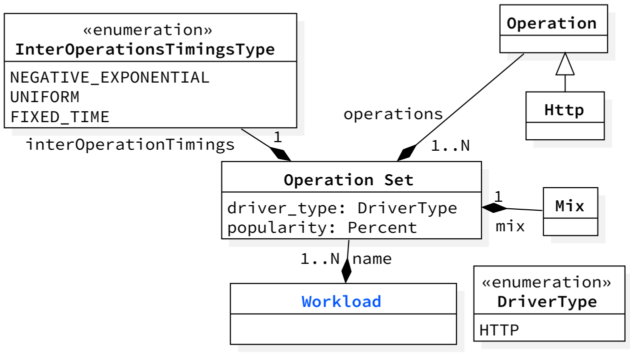

Figure 7: **DSL: The Workload Meta-Model**

the operations and the `inter_operation_timings`. We rely on Faban[13] and its driver meta-model as performance test execution framework (Sect. 4), and we expose in the DSL all the supported mixes of operations and `inter_operation_timings` Faban supports. These are: `fixed-sequence` defining a fixed order over the operations; `flat-mix` randomly deciding the next operation based on the corresponding probabilities; and `flat-sequence-mix` a combination of `fixed-sequence` and `flat-mix` that allows the user to specify a random selection of fixed sequences, as well as `matrix-mix` that implements a Markov-chain model and select the next operation based on the current operation and the provided probability. More details are available on the Faban documentation, and we omit them here for space reason. The supported `inter_operation_timings` are: `negative-exponential`, `uniform` or `fixed-time`. The `inter_operation_timings` also require configurations, omitted because not central for this work.

Sut - By relying on the integration of the DSL runtime with Docker technologies, the user can also control the SUT configuration and deployment in a declarative way as modeled in Fig. 8. The deployment descriptor of the SUT is currently specified using the Docker Cloud and Compose Standard, and in the DSL is possible to override some resource settings and configurations, and decide which services of the SUT should be deployed on which server (identified using an alias), other than specify a name and a version for the SUT. This way it is possible to reuse the deployment descriptor across different tests.

The user can also decide which service is the target of the defined test and its endpoint, and how to determine that the targeted service is ready to accept requests (currently a regular expression matched against the target service logs). 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.

[13] http://faban.org, last visited April 11, 2018

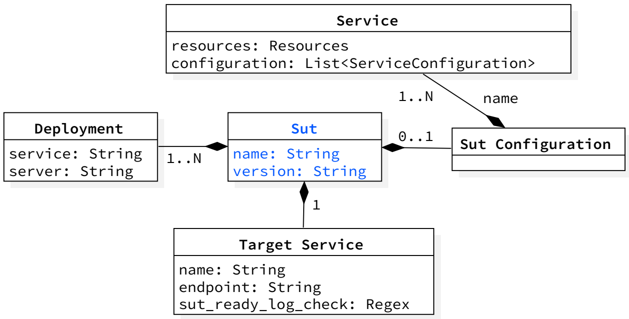

Figure 8: **DSL: The SUT Meta-Model**

Data Collection - In order 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. On the client-side we rely on Faban to collect workload performance metrics, thus a Faban collector service can be specified, and optionally configured. 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 data stored in databases. The list can be easily extended by integrating new data collector services, according to the user's needs. For many data collector services 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). To be able to collect all the data required to respond to current and future users need, by default all collector that do not require configurations are enabled on all the services.

3.3 DSL Library and Static Validation

The DSL has been designed to be integrated in CSD, and other than enabling the language with specific entities related to the context, we also make sure that the meta-model behind the DSL can be utilized within those contexts. In CSD different tools are employed to build and control the development and deployment of systems [9], thus being able to access the language features within those tools is fundamental for extensibility. For this reason, we implemented a library exposing the DSL meta-model in a functional way to other programs, so that other systems can parse a DSL definition serialized in YAML, or can rely on a Builder interface to create an instance of the meta-model to be submitted to the framework. By leveraging the meta-model presented in Sect. 3, the library ensures the definition is syntactically and semantically valid, before an instance of a test can be instantiated. This is very important in the context of performance testing and CSD, 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 can be spotted early. The meta-model definition supports by design syntactic validation, by ensuring entities can be parsed only if correctly structured and if using the data types we enforce in the model. For what concern semantic validation, we ensure that the definition of the test is consistent, by verifying that each 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).

4 CONTINUOUS, END-TO-END PERFORMANCE TEST AUTOMATION

The DSL, and its library, are paired with a model-driven framework which drives the end-to-end lifecycle of the performance test execution embedded into a continuous software delivery process, presented in this section.

4.1 End-to-end Performance Test Automation

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, issuing the workload, deploying the SUT, collecting client- and server-side data, undeploying the SUT and data analysis. The overall process of end-to-end performance test execution automation as described with the DSL consists of three phases: exploration, execution and analysis.

Exploration - The exploration phase handles the way a performance test is executed in order to reach its goal, and it is the central phase to the lifecycle. Before the test starts, following the test definition described in the DSL, the exploration phase handles the necessary preparations before the performance test is executed in order to reach the goal specified by the user. After the test definition has been statically verified for correctness, based on the goal, one or more experiment definitions are generated together with the corresponding SUT deployment descriptor, and a given number of trials are scheduled for execution.

Given its central role, after the execution of each experiment trial, the exploration phase is also in charge of taking action based on the results of the analysis phase. The exploration phase receives all the metrics declared in the `observe` DSL entity, so that termination criteria and quality gates can be evaluated to decide how to continue the exploration, or failures data about something that went wrong during the execution (Sect. 4.3).

Execution - Deploying and running a performance test is done in the execution phase. The first step of the execution phase concerns the SUT deployment. To deploy the SUT we use Docker[14] containers - a lightweight virtualization platform. The SUT deployment descriptor includes both configuration environment variables and resource constraints, and by altering the specification we are able to automatically define performance tests that involve system and resource

[14] Docker - https://www.docker.com, last visited April 11, 2018

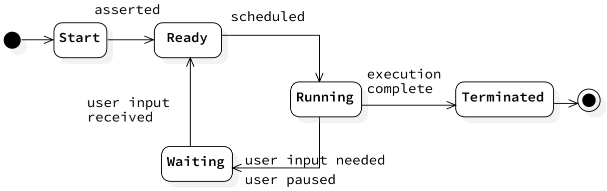

Figure 9: **Test Lifecycle - High-Level States**

configuration. Before starting each trial, a new deployment of the SUT is performed, to ensure that the test starting conditions are always the same for all the trials. After deployment, the experiment execution starts according to the defined workload. The framework takes also care of starting the data collection services needed to collect performance data relevant for the test.

Analysis - In the analysis phase performance data produced by the experiments is collected and analyzed. The computed metrics and statistics are provided to the users for analysis, and fed back in the exploration phase so that decisions on how to continue the exploration can be made.

In this work we mainly focus on the automation of the exploration phase, thus more details are provided on this phase in Sect. 4.2 and Sect. 4.3. For the automation of the execution and the analysis phases, and the overall architecture of the framework, one could refer to our previous work [7].

4.2 Performance Test Exploration Lifecycle

The exploration phase is at the core of declarative goal-driven performance test automation execution. To drive the automated execution, goal exploration, and failures handling of performance tests, we defined a lifecycle implemented through a state machine. The complete lifecycle is realized by a test lifecycle, driving the exploration of the goal, and an experiment lifecycle, driving the execution of the different trials of a single experiment that is scheduled.

Fig. 9 depicts the high-level states of the test lifecycle. The high-level states are inspired by the scheduling of processes in an operating system: `start` (new), `ready`, `running`, `waiting` and `terminated`. The start state is reached after the test has been verified as syntactically and semantically correct, and setup the framework to be ready for test execution (i.e., stores relevant data to be accessible in next states). 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 framework. If errors are encountered, the stored data are deleted, and the user is alerted that the test can not be scheduled for execution. When there are resource available for execution, the test is moved from the ready state to the running state, where the actual execution of the test exploration and experiment execution happen. The user, or the system, could decide to pause the test. In this case the test is moved to the waiting state, waiting for user input before proceeding. Once the execution of the test is completed, it is moved to the terminated state, that represent a final test state after which the state of the test can change only if re-started by the user. The running state is the most rich and complex one, thus we define different substates, that are presented in Fig. 10. The same figure also reports the possible final terminated substates a test can end in. The `running` state is divided into several substates: `determine exploration strategy`, `determine and execute experiments`, `handle experiment result`, `validate termination criteria`, `check quality gates`, and `terminating`. The different states of the state machine, are mapped to the DSL entities, and represent the execution semantics driven by the declarative specification described in Sect. 3. The `determine exploration strategy` state is used to set a predefined *exploration strategy* for experiment selection in case the user did not specify one in the test. In the `determine and execute experiments` the experiment to be executed next is determined by the selection strategy. After the selection, the control is handed over to the experiment lifecycle, that we do not show for space reason. The experiment lifecycle handles the execution of trials, and re-execution of the same in case of failures, as described in Sect. 4.3. As per the test lifecycle, also for the experiment lifecycle, termination criteria defined in the test definition are verified, after each trial complete the execution. The possible final sub states for the experiment *terminated* state are: *completed*, when the execution complete as expected, *error* if an error incurs in generating the experiment bundle or in setting up the experiment on Faban for execution, *failure* when some termination criteria can not be reached before the end of the execution (e.g., the expected confidence interval at the wanted precision can not be reached within the defined maximum number of trials) or some unexpected failures happen during the execution (Sect. 4.3), or *aborted* if the execution is aborted due to reaching time based termination criteria in the test lifecycle. Once the experiment has executed, the SUT undeployed, performance data collected and the result, e.g., execution status and metrics, of the execution is available, 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. If the data is not received within a given maximum time, e.g., due to errors in data analysis, then the lifecycle is

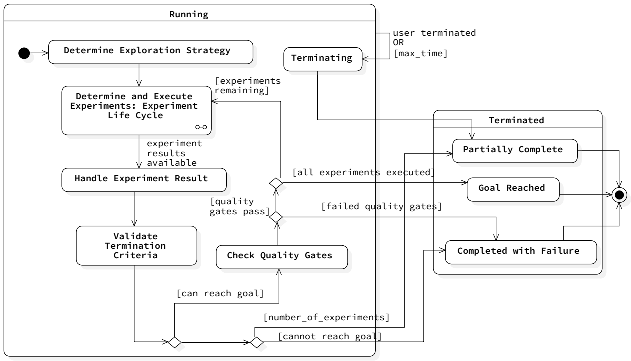

Figure 10: **Test Lifecycle - Running and Terminated States**

moved to the next state that evaluates the absence of data and ends the test as *partially completed*. After the result is collected, the termination criteria are verified in the `validate termination criteria` state. If the termination criteria verification stops the experiment before the wanted number of experiment is executed, then according to the actual termination criteria the final terminated inner state is *partially complete* or *completed with failure*. The *partially complete* state can also be reached if the user decides to abort the execution or if a time based termination criteria is triggered. If the termination criteria do not stop the execution, then the quality gates are evaluated. If the quality gates pass, then the next experiment, if present, is executed, or the test is moved into the *goal reached* state. If the quality gates fail, the test is moved in the *completed with failure* state and the reason of the failure is embedded as part of the state so that it can be reported to the user.

4.3 Failures Handling

The end-to-end performance test execution can incur in several failures that are unexpected, even though we make sure to statically verify that the test definition in the DSL is syntactically and semantically correct. As performance tests often require a significant amount of execution time, automatic failure handling is crucial in CSD, and failures should therefore be handled as soon as possible. We defined a taxonomy of possible runtime errors that we could encounter during the execution.

The taxonomy covers all the cases we were able to identify, even though it can not yet be considered exhaustive. The taxonomy differentiates between failures that can incur on the three different levels: test level, experiment level and trial level. On the test level a failure happens when the data provided by the results of an experiment are not sufficient to reach the goal (e.g., because the experiments can not be successfully executed), and therefore the test is terminated prematurely. In the cases where a failure on the experiment level does not impact the overall goal, the framework on the other hand continue with the exploration. Experiment level failures are caused by trial failures and stops the execution of additional trials for the given experiment since these are also expected to fail. The trial failures are directly related to the execution of a performance test. It can either be a failure that directly causes an experiment failure since additional executions would likely cause the same behavior (e.g., wrong specified endpoint, or fatal errors in deploying and verifying the SUT as ready), or the trial can be re-executed (if the re-execution does not succeed after a given number or retries, then an experiment failure is triggered) if the failure was more of a random nature. Currently, the presence of failures in verified after each trial completes its execution.

4.4 Open-source Framework as a Service

The framework, as the DSL, has been designed to be integrated in CSD environments. It is open-source[15], extensible by design and deployed as a service exposing REST APIs. The framework is meant to be used by users for exploratory testing and for automated execution of tests in CSD. Users might want to execute tests at any moment in time, and for this we provide a command line client interacting with the APIs. When tests are automated in CSD, another entry point for test scheduling are other tools in the CSD lifecycle, such as continuous integration systems. These systems can interact with the framework using the same REST APIs as the command line client, and issue pre-defined performance test definitions serialized in YAML, or performance tests built on the fly using the DSL library given the current needs of the CSD process.

The framework is developed to take performance test execution requests, and automate the process of execution up to the point in which users input is required (i.e., when results are ready or failures happen in the process). When executing performance tests, especially in CSD, traceability and reproducibility of what happens is very important, also to improve the process. For this reason the framework logs all the steps and decisions taken during the test execution, such that the process is transparent and inspectable by the users.

5 USE CASES

In this section we present different real-world use cases from our experience in applying the framework in different research contexts. The use cases are meant to show: 1) how the declarative approach implemented in the DSL, and its expressiveness, enables different performance testing activities (use cases: Load Test, Exhaustive Exploration Test, Stability Boundary Test); 2) how the automation framework is configured using the DSL's meta-model, and control the end-to-end lifecycle of test executions (use cases: Termination Criteria, Quality Gates).

We omit final results showing the outcome of the use case execution fro space reason, as the focus of the paper is on the declarative performance test definition using the DSL, and the test execution by the framework.

Load Test - This use case shows how the user can specify a load test using the provided DSL, that for example could be set to be executed continuously as part of nightly builds of the SUT. The SUT we refer to in this use case is realized by two services, one named *catalogue_ws* and a second one named *dbms*. The services are connected and represent a REST Web service handling an items catalogue and the DBMS it relies on. The user's intent is to issue a load test, with the defined load function, and observe some defined metrics, under a given SUT configuration provided using the model presented in Fig. 8, and a given workload where the simulated users interacting with the SUT browse the catalogue, that we omit for space reason. The user is interested in observing the response time metrics for the browse workload, as well as metrics related to the RAM and CPU utilization for both of the services. Listing 2 presents the YAML format for the test specification, with omitted details that are not central to the use case. When the framework executes a load

[15] The BenchFlow framework GitHub repository - https://github.com/benchflow, last visited April 11, 2018

```
configuration:
 goal:
  type: load_test
  observe:
   workload:
    browse: [response_time]
   services:
    catalogue_ws: [ram, cpu]
    dbms: [ram, cpu]
  load_function:
   users: 1000
   ramp-up: 2m
   steady-state: 10m
   ramp-down: 2m
```

Listing 2: **Load Test: Metrics and Load Function**

test, only one experiment is executed by the lifecycle presented in Fig. 10, thus after the experiment terminates the execution, the test is concluded. In the case of this example, since no termination criteria nor quality gates are defined, the test ends up in the *goal reached* state, unless failures happen during the execution (for which 3 trials are scheduled by default) or in retrieving the test result, in which cases the final state would be *completed with failure*. The framework implements some default failure handling mechanism, as presented in Sect. 4.3, so that to avoid wasting resources to execute tests that can not reach the final goal. If the test can not reach the final goal its execution is stopped, independently of eventually defined termination criteria or quality gates. After a successful execution, the user is provided with access to the metrics declared in the *observe* section of the test definition and relevant statistical tests to help the user investigate the quality of the obtained results. The user can also optionally access artifacts generated during the automation process, and all the collected raw data, for transparency and reproducibility.

Exhaustive Exploration Test - In this use case we show how a user can define an exploratory test, that performs an exhaustive exploration of the described performance space.

The user's intent could be to learn about the performance of the developed system when setting different configurations and resource allocations. Listing 3 presents the YAML specification responding to this goal, where we omit the specification of the *load function* and the *observe*d metrics, that are the same as in Listing 2. By relying on the declarative approach provided in the DSL, the user specifies that she wants to explore the performance in all the 48 configurations in the exploration space defined by the Cartesian product of all the values of the specified exploration variables: *NUM_SERVICE_THREAD* as configuration setting of the catalogue_ws service, *memory*, and CPU as resource settings for the dbms service. As shown in Listing 3 the user can rely on a *step* based definition, as she is doing for exploring the *memory*, to define the way to determine the values to explore in the exploration space, that in the case of the example includes all the values between 2GB and 24GB with a step of 2GB.

```
configuration:
  goal:
   type: exhaustive_exploration
   exploration:
    exploration_space:
     services:
      catalogue_ws:
       configuration:
        NUM_SERVICE_THREAD: [12, 24]
      dbms:
       resources:
        memory:
         range: [2GB, 24GB]
         step: +2GB
        cpu: [4, 8]
    exploration_strategy:
     selection: one-at-a-time
```

Listing 3: **Exhaustive Exploration Test: Exploration Space**

When the goal declared in Listing 3 is executed by the lifecycle presented in Fig. 10, all the experiments in the exploration space are scheduled one after the other, following the *one-at-a-time* selection strategy that executes the experiments in the order they are defined in the exploration space. In this use case we can see how simple would be to change the way experiments are selected, to explore the space in different ways and getting first results that belongs to other regions of the space. If a user would like to do so, it is a matter of changing the value of the *exploration_strategy.selection* setting to the other strategies that are made available by the framework, or custom strategies added by the user. As in the load test use case, since no termination criteria nor quality gates are specified, if there are no failures the test terminates in the *goal reached* state after all the experiments have been executed, otherwise it terminates in the *terminated with failure* state.

Termination Criteria - Given the expressiveness of the proposed declarative DSL, the user can define tests executing many experiments and potentially lasting a long amount of time to be completed, and that can incur into errors. In this use case we show how a user can have control on the execution lifecycle of the test defined in Listing 3, to set conditions leading to a premature termination of the test execution. This way the users can rely on the automation provided by the framework to continuously execute tests, and she has control on the time allocated to their execution and in deciding when avoiding wasting resources because the test can not be considered valid. In Listing 4 the user decides to define a maximum runtime for the test of 20 hours, with a maximum number of failed experiments set to 5% of the total number of experiments to schedule (48). Each experiment is set to be

```
configuration:
  termination_criteria:
    test:
      max_time: 20h
      max_failed_experiments: 5%
    experiment:
      workload:
        browse:
          confidence_interval_metric:
            ↪  avg_response_time
          confidence_interval_value: 50ms
          confidence_interval_precision: 95%
          max_number_of_trials: 10
```

Listing 4: **Exhaustive Exploration Test: Termination Criteria**

executed multiple times, with a dynamic termination criteria set on the *browse* workload. The termination criteria states that the confidence interval of the *avg_response_time* has to be 50ms at 95% of confidence level, with an upper limit of 10 trials.

When the framework executes the declared goal, each experiment is repeated a variable number of times, and if the wanted confidence interval is achieved within the *max_number_of_trials* its execution is marked as successful. On the contrary, if fatal errors happens, or the confidence interval can not be obtained within 10 trials, the experiment execution is marked as failed. The test is executed for a maximum time of 20 hours, and if it is not completed within the maximum time, it is suspended and moved to the *partially complete* terminated state from where the user could decide to ask the framework to continue its execution by extending the amount of time it can run. If during the execution more than 5% of the experiments fail because of one of the failures presented in Sect. 4.3 or because termination criteria are failing the test, then the test is moved to the *completed with failure* state.

As for the other user cases, the user can access all the metrics and produced data, that in this case would also contain data related to eventual failures that happened.

Quality Gates - The use case presented in Listing 5 is also based on the one presented in Listing 3, and in this case the user's intent during the exploration of the performance of the system in the specified space, is verifying if the system does not exceed a specified $95th$ percentile for the registered response time for the entire browse workload, that the maximum deviation from the specified mix is less or equal to 2%, and that the average CPU utilization of the *catalogue_ws* is less or equal than 70%. By setting quality gates, the user has control on the final result of the test, and can decide to stop the execution as soon as these quality gates are not achieved, marking the test as failed, or continue its execution reaching a successful state. This is important because often, in CSD environments, continuous validation of performance benchmarks are executed, to continuously verify that the system keeps achieving specified quality conditions in given configurations.

```
configuration:
   quality_gates:
     workload:
       browse:
         95thp_response_time: <= 250ms
         max_mix_deviation: 2%
     services:
       catalogue_ws:
         avg_cpu: <= 70%
```

Listing 5: **Exhaustive Exploration Test: Quality Gates**

```
configuration:
    goal:
      type: stability_boundary
      exploration:
        stability_criteria:
          workload:
            browse:
              max_mix_deviation: 5%
          services:
            catalogue_ws:
              avg_cpu: <= 80%
            dbms:
              avg_cpu: <= 90%
        exploration_strategy:
          selection: stability_boundary_first
```

Listing 6: **Stability Boundary Test**

Stability Boundary Test - In this last use case, we show how the user can specify a more advanced goal, namely a stability boundary goal. We base this use case on the Listing 3, and in Listing 6 we present the changes to be applied to the specification in order to define a stability boundary goal. In the case of stability boundary goal, the *stability_boundary_first* selection strategy has to be specified, and a new section has to be added in the specification, namely *stability_criteria*, to define the stability criteria for the SUT. The *stability_boundary_first* selection strategy is enabled to use the defined stability criteria and decide the order of experiments to be executed in the defined exploration space. In the case of Listing 6 the user is setting stability criteria on the workload and services average CPU utilization.

The framework, by applying the *stability_boundary_first* strategy, determines the order of the experiments such that the first to be executed are the ones were the system is expected to be less stable, and then applies a binary search in the exploration space to trace the stability boundary, if the system is not stable in the first set of mentioned explored points. The current assumption is that the system 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 as it is in the case of Listing 3). The idea is that in this way the user can start to collect data about regions of the exploration space were the system is more likely to be not stable, and although the space could be explored completely in case the system is stable in the entire exploration space, she could set termination criteria based on time or on maximum number of failed experiments to decide when to stop the exploration.

6 CONCLUSION AND FUTURE WORK

In this work we presented a declarative DSL for specifying goal-oriented performance tests, mainly focused on container-packaged (Micro)service systems, and a model-driven framework that enables the automated end-to-end execution of the specified tests. As shown with different use cases, the DSL allows developers to explicitly declare the goal of the performance test, as well as precisely control the deployment configuration of the SUT. The framework automates the end-to-end execution of performance tests defined using the DSL, whose execution semantics is defined in terms of state machines controlling the tests execution and allowing to statically checking tests for correctness. The declarative nature of the DSL makes explicit the intent, purpose and applicability context of the defined tests, as well as opening up the possibility to use alternative exploration strategies to achieve a given goal. Both the DSL and the framework have features relevant for CSD environments, such as termination criteria, quality gates, observable metrics and user-defined failure handling mechanisms. As future work we plan to extend the set of goals supported by the DSL, add additional strategies to navigate the exploration space, and integrate statistical model based techniques, to speed up the exploration space exploration [21]. We also plan to collect structured users' feedback on the declarative DSL after applying it to more real-world usage scenarios.

ACKNOWLEDGMENTS

This work is funded by the "BenchFlow" project (DACH Grant Nr. 200021E-145062/1) project.

REFERENCES

[1] Varsha Apte, T V S Viswanath, Devidas Gawali, Akhilesh Kommireddy, and Anshul Gupta. 2017. AutoPerf: Automated load testing and resource usage profiling of multi-tier internet applications. In *Proc. of ICPE*. 115–126.

[2] Maicon Bernardino, Avelino F Zorzo, and Elder M Rodrigues. 2014. Canopus: A Domain-Specific Language for Modeling Performance Testing. In *Proc. of ICSEA*. 157–167.

[3] Andreas Brunnert and Helmut Krcmar. 2017. Continuous performance evaluation and capacity planning using resource profiles for enterprise applications. *Journal of Systems and Software* 123, 1 (2017), 239–262.

[4] Andreas Brunnert, André van Hoorn, Felix Willnecker, et al. 2015. *Performance-oriented DevOps: A Research Agenda*. Technical Report. SPEC RG DevOps.

[5] Matheus Cunha, Nabor C Mendonça, and Américo Sampaio. 2013. A Declarative Environment for Automatic Performance Evaluation in IaaS Clouds. In *Proc. of CLOUD*. 285–292.

[6] Stefano Di Alesio, Shiva Nejati, Arnaud Gotlieb, and Lionel Briand. 2013. Stress testing of task deadlines - A constraint programming approach. In *Proc of. ISSRE*. 158–167.

[7] Vincenzo Ferme, Ana Ivanchikj, and Cesare Pautasso. 2015. A Framework for Benchmarking BPMN 2.0 Workflow Management Systems. In *Proc. of BPM*. 251–259.

[8] Vincenzo Ferme and Cesare Pautasso. 2017. Towards Holistic Continuous Software Performance Assessment. In *Proc. of ICPE Companion*. 159–164.

[9] Brian Fitzgerald and Klaas-Jan Stol. 2017. Continuous software engineering: A roadmap and agenda. *Journal of Systems and Software* 123, 1 (2017), 176–189.

[10] Ilias Gerostathopoulos, Tomas Bures, Sanny Schmid, Vojtech Horký, Christian Prehofer, and Petr Tůma. 2016. Towards Systematic Live Experimentation in Software-Intensive Systems of Systems. In *Proc. of ECSA*. 1–7.

[11] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, and D Sculley. 2017. Google Vizier - A Service for Black-Box Optimization. In *Proc. of KDD*. 1487–1495.

[12] Juha Itkonen, Casper Lassenius, and Eero Laukkanen. 2017. Problems, causes and solutions when adopting continuous delivery - A systematic literature review. *Information and Software Technology* 82, 2 (2017), 55–79.

[13] Nicolas Michael, Nitin Ramannavar, Yixiao Shen, Sheetal Patil, and Jan-Lung Sung. 2017. CloudPerf - A Performance Test Framework for Distributed and Dynamic Multi-Tenant Environments. In *Proc. of ICPE*. 189–200.

[14] Jean Christophe Petkovich, A Oliveira, Y Zhang, Thomas Reidemeister, and Sebastian Fischmeister. 2015. DataMill: a distributed heterogeneous infrastructure for robust experimentation. *Software: Practice and Experience* 46, 10 (2015), 1411–1440.

[15] A Omar Portillo-Dominguez, Miao Wang, John Murphy, Damien Magoni, Nick Mitchell, Peter F Sweeney, and Erik Altman. 2014. Towards an automated approach to use expert systems in the performance testing of distributed systems. In *Proc. of JAMAICA*. 22–27.

[16] Gerald Schermann, Dominik Schöni, Philipp Leitner, and Harald C Gall. 2016. Bifrost - Supporting Continuous Deployment with Automated Enactment of Multi-Phase Live Testing Strategies. In *Proc. of Middleware*. 12:1–12:14.

[17] Joel Scheuner, Philipp Leitner, Jürgen Cito, and Harald C Gall. 2014. Cloud Work Bench - Infrastructure-as-Code Based Cloud Benchmarking.. In *Proc. of CloudCom*. 246–253.

[18] Simon Spinner, Giuliano Casale, Fabian Brosig, and Samuel Kounev. 2015. Evaluating approaches to resource demand estimation. *Performance Evaluation* 92, C (2015), 51–71.

[19] Jan Waller, Nils C Ehmke, and Wilhelm Hasselbring. 2015. Including Performance Benchmarks into Continuous Integration to Enable DevOps. *ACM SIGSOFT Software Engineering Notes* 40, 2 (2015), 1–4.

[20] Jürgen Walter, André van Hoorn, Heiko Koziolek, Dusan Okanovic, and Samuel Kounev. 2016. Asking "What"?, Automating the "How"? - The Vision of Declarative Performance Engineering. In *Proc. of ICPE*. 91–94.

[21] Dennis Westermann. 2014. *Deriving Goal-oriented Performance Models by Systematic Experimentation.* Ph.D. Dissertation. Karlsruhe Institute of Technology.

[22] Johannes Wettinger, Uwe Breitenbücher, Michael Falkenthal, and Frank Leymann. 2016. Collaborative gathering and continuous delivery of DevOps solutions through repositories. *Computer Science - Research and Development* 31, 4 (2016), 1–10.

[23] Liming Zhu, Len Bass, and George Champlin-Scharff. 2016. DevOps and Its Practices. *IEEE Software* 33, 3 (2016), 32–34.