Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Automated test-based learning and verification of performance models for microservices systems[☆]

Matteo Camilli[*], Andrea Janes, Barbara Russo

Faculty of Computer Science, Free University of Bozen-Bolzano, Piazza Domenicani 3, 39100 Bolzano, Italy

ARTICLE INFO

Article history:
Received 1 April 2021
Received in revised form 2 December 2021
Accepted 12 January 2022
Available online 26 January 2022

Keywords:
Performance testing
Test-based model learning
Markov models
Automated verification
Microservices

ABSTRACT

Effective and automated verification techniques able to provide assurances of performance and scalability are highly demanded in the context of microservices systems. In this paper, we introduce a methodology that applies specification-driven load testing to learn the behavior of the target microservices system under multiple deployment configurations. Testing is driven by realistic workload conditions sampled in production. The sampling produces a formal description of the users' behavior through a Discrete Time Markov Chain. This model drives multiple load testing sessions that query the system under test and feed a Bayesian inference process which incrementally refines the initial model to obtain a complete specification from run-time evidence as a Continuous Time Markov Chain. The complete specification is then used to conduct automated verification by using probabilistic model checking and to compute a configuration score that evaluates alternative deployment options. This paper introduces the methodology, its theoretical foundation, and the toolchain we developed to automate it. Our empirical evaluation shows its applicability, benefits, and costs on a representative microservices system benchmark. We show that the methodology detects performance issues, traces them back to system-level requirements, and, thanks to the configuration score, provides engineers with insights on deployment options. The comparison between our approach and a selected state-of-the-art baseline shows that we are able to reduce the cost up to 73% in terms of number of tests. The verification stage requires negligible execution time and memory consumption. We observed that the verification of 360 system-level requirements took $\sim$1 minute by consuming at most 34 KB. The computation of the score involved the verification of $\sim$7k (automatically generated) properties verified in $\sim$72 seconds using at most $\sim$50 KB.

1. Introduction

The term "microservices" is used to denote applications built as suites of loosely coupled specialized services, each running in its process and communicating with lightweight mechanisms, such as an HTTP resource API (Dragoni et al., 2017). This architectural style lends itself to decentralization and to the adoption of continuous integration and deployment practices, as reported by several companies (e.g., Amazon and Netflix) that successfully developed and deployed microservices applications in their production environment. Several configuration alternatives are possible for microservice deployment, for example, serverless microservices using functions (e.g., Amazon Lambda[1]), container-based deployment (e.g., Docker[2]), either physical or virtual machines, or a combination of them with different hardware capacity constraints. Companies report that introducing a microservices architecture often adds more communication between the different services (Alshuqayran et al., 2016). This often affects performance (i.e., how the target microservices system performs upon user requests) and scalability (i.e., how the target microservices system performs when increasing the scale of operation) that represent fundamental quality attributes that shall be assured with systematic methods supporting the engineering life-cycle (Soldani et al., 2018; Avritzer et al., 2020). As described in Soldani et al. (2018), performance testing of microservices is recognized as challenging in the IT industry that increasingly integrates development and operations in an overall framework referred to as DevOps (Bertolino et al.). As described in Ghezzi (2018), in this context there is the need of methods that can be practiced, and systematically replicated. Mainstream approaches to

☆ Editor: Raffaela Mirandola.
* Corresponding author.
E-mail addresses: matteo.camilli@unibz.it (M. Camilli), andrea.janes@unibz.it (A. Janes), barbara.russo@unibz.it (B. Russo).
1 https://aws.amazon.com/lambda/
2 https://www.docker.com/

https://doi.org/10.1016/j.jss.2022.111225

performance assessment in industry focus on the passive monitoring of the system response time or resources utilization to detect anomalous performance and scalability issues, such as bottlenecks (Heinrich et al., 2017). Even approaches based on load or stress testing, usually extract a set of performance indices and statistics that are difficult to use for guiding engineering decisions because of missing connections with system-level requirements (Jiang and Hassan, 2015; Ferme and Pautasso, 2018).

The main goal of this work is to address the aforementioned limitations by enabling automated decision gates in performance testing of microservices that allow requirements traceability. We seek to achieve this goal by endowing common performance testing practices used in microservices systems, with the ability to automatically learn and then formally verify a performance model of the System Under Test (SUT) to achieve strong assurances of quality.

To achieve our main goal, in this paper we introduce a methodology that integrates load testing and *Bayesian* inference (Robert, 2007) to learn a formal model of the SUT and then *probabilistic model checking* (Kwiatkowska et al., 2011) to automatically verify system-level requirements after accounting for runtime evidence. More in detail, our methodology has the following steps: (*i*) design-time analysis of the target operational setting; (*ii*) performance model learning via load testing and inference; then (*iii*) verification of requirements, and computation of the configuration score.

The design-time *analysis* phase aims at defining the behavior of multiple user categories and the *workload intensity* (i.e., arrival rate of concurrent users) that are likely to occur in production by studying the operational data. User categories are then used to build a formal description of the usage profile in terms of a Discrete Time Markov Chain (DTMC) (Norris, 1998). The workload intensity, the alternative deployment configurations, and the usage profile collectively compose the input of the *learning* activity. This stage executes multiple load testing sessions for each target operational setting. Each session loads the SUT following the DTMC model and feeds a Bayesian inference process that incrementally learns performance parameters based on the collected runtime evidence. The outcome is a set of Continuous Time Markov Chains (CTMC) (Anderson, 2012), one for each deployment configuration, that describes the stochastic behavior of user categories as well as performance rates associated with microservices. Models are then *verified* in the last stage against system-level requirements using the probabilistic model checker PRISM (Kwiatkowska et al., 2011). Furthermore, we use the model checker to automatically compute a score that evaluates each individual deployment configuration based on the ability to fit the operational setting.

In this paper, we present a comprehensive discussion of theoretical aspects related to the core stages of our methodology and we describe the current toolchain implementation we used to conduct automated experiments in our in-vitro testing environment. Specifically, we conducted an experimental campaign to evaluate our approach using a representative microservices system called "Sock Shop" with different configurations based on alternative memory allocations, amount of CPU, and number of microservice replicas. Results of our controlled experiments show that the learning process provides the ability to trace performance issues back to violated requirements. Violations can be detected with limited effort. We show that our approach is cheaper compared to a selected state-of-the-art baseline called Domain Metric (Avritzer et al., 2020) that has been recently applied to evaluate the performance of microservices systems in different industrial domains (Avritzer et al., 2021b,a). In particular, the rigorous foundation of our approach reduces the cost up to 73% in terms of number of tests required by the load testing sessions. The verification of the inferred models requires negligible execution time and memory consumption. We observed that the verification of 360 system-level requirements took $\sim$1 min by consuming at most 34 KB. The computation of the score involved the verification of $\sim$ 7k (automatically generated) properties verified in $\sim$ 72 s using at most $\sim$ 50 KB. The outcome of our evaluation shows also that the choice of the deployment configuration that better fits the operational setting is not trivial. Thus, our approach provides engineers with suitable metrics to drive the identification of problematic deployment configurations.

The key contributions are as follows:

- A novel methodology for model learning and verification of microservice-based systems under different deployment alternatives, via automated load testing, Bayesian inference, and probabilistic model checking.
- The evaluation of our methodology on a representative microservice-based system benchmark pointing out benefits, costs, and threats to validity.

The remainder of this paper is structured as follows. In Section 2 we introduce a preview of our approach. In Section 3 we recall the necessary background concepts. Our target benchmarking system of choice (i.e., Sock Shop), used throughout the article to illustrate the main phases of the methodology, is presented in Section 4. In Section 5 we introduce a rigorous and comprehensive treatment of our model learning and verification methodology. In Section 6 we report the evaluation carried out to assess the benefits and the costs of the core stages of our methodology. In Section 7 we describe related work. In Section 8 we discuss the main strengths and limitations of our approach. Finally, in Section 9 we conclude the paper and we discuss future directions of this work.

2. Preview of the approach

As anticipated in Section 1, we focus on microservices systems modeled as CTMC and quantitative requirements expressed using the Continuous Stochastic Logic (CSL) notation (Forejt et al., 2011). As described in Filieri et al. (2016), Markov models such as DTMC and CTMC represent widely accepted formalisms to model and verify software system reliability and performance (Filieri et al., 2012; Guerriero et al., 2019). We assume that the rate associated with model transitions is initially unknown/uncertain. In particular, nonfunctional aspects (e.g., response time of microservices) are hard to predict during the early design stages and they may even change depending on environmental conditions like the operational profile or the system configuration.

Fig. 1 illustrates a high-level schema of our approach. The preliminary stage concerns the (*i*) *analysis of the operational setting* whose aim is to define the behavior of the users and study the operational data to extract a distribution of the workload intensity that is likely to occur in production. The distribution provides our methodology with the workload intensities representing representative conditions that need to be covered during testing. A number of user categories can be manually defined by the tester or automatically extracted from the operational data usually collected by observing the system in production in a DevOps setting (Bertolino et al., 2020). Each category is specified as a valid interaction workflow in terms of sequences of requests to microservices. Starting from this definition, we automatically build a model of the usage profile in terms of a DTMC model. According to the analysis stage, the DTMC specifies the stochastic behavior of users that is likely to occur in production. Then, the distribution of the workload intensity and the DTMC model are given as input to the model learning.

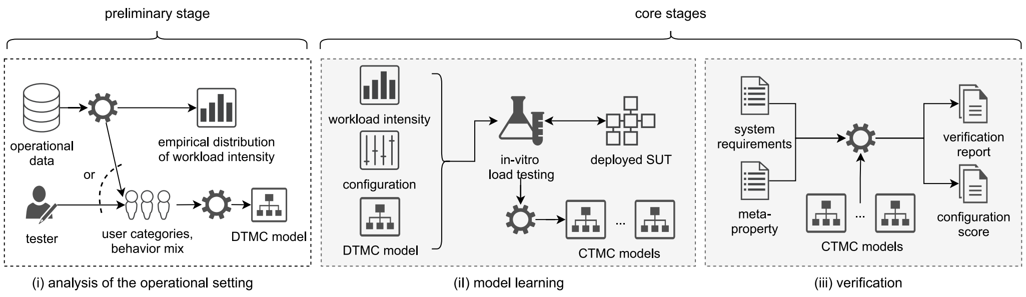


Fig. 1. High-level schema of the test-based model learning and verification methodology. Gear icons denote automated machinery steps, whereas arrows indicate the data flow.

In the (*ii*) *model learning* stage, the tester specifies the prior knowledge (if any) on the SUT behavior in terms of performance rates. Then, we automatically execute multiple load testing sessions to query the SUT and incrementally learn a full specification for the system behavior. For each representative workload intensity and available deployment configuration, the SUT is loaded by generating synthetic users following the DTMC model. The learning process leverages *Bayesian inference* to incrementally update the partial specification (i.e., the DTMC model) by augmenting it based on the collected runtime evidence. This process produces as an outcome a CTMC model encoding the full specification learned for each workload intensity and deployment configuration. The CTMC models learned during the testing sessions compose the input of the (*iii*) *verification* stage. Here we leverage the probabilistic model checker PRISM to verify the learned models under different operational settings and deployment configurations. The outcome of the model checker can be used, for instance, by engineers as readiness criteria for a new release of the system. In addition to verification of requirements, we introduce the notion of *configuration score* to rank the available configurations and select the one that is likely to best fit the target operational setting to support engineers during the final "go live" decision gate.

It is worth noting that our approach does not focus on estimating average behavior. Indeed, especially under stress, the response time of microservices tends to result in heavy-tailed distributions (typically due to higher resource contention or accumulation of requests in processing queues). In these cases, a large portion of requests often falls in the tail, skewing the means of the distributions. Therefore, requirements evaluated on the means may fail to capture the quality of service perceived by the users. For instance, a service with a mean of 200 milliseconds may still serve 10% of the users in more than 1 s. Thus, rather than focusing on average behavior, we infer the full probability density functions that we encode as parameters of a CTMC model.

It is worth noting that probabilistic model checking (in the last phase) requires exhaustive exploration of the model's state space to analyze arbitrarily complex CSL properties (Courcoubetis and Yannakakis, 1995; Clarke et al., 2000). We keep this activity separated from model learning as an offline phase, where we can execute demanding activities without interfering with the system operation. The aim here is to make the online stages of our methodology as lightweight as possible. In particular, the learning process collects evidence and executes Bayesian inference which is computationally inexpensive, as described in the next sections.

3. Background

This section recalls the background notions needed to formalize our approach. We briefly revisit DTMC and CTMC models, the quantitative temporal logic CSL, and Bayesian Inference. We refer the reader to the bibliography reported below for a complete treatment of these topics.

3.1. Continuous Time Markov Chains

CTMCs (Anderson, 2012) are continuous time stochastic processes with discrete state space. The modeled system remains a certain amount of time (following a non-negative exponential distribution) at each state and then it evolves according to probabilities that depend only on the leaving state. Formally, a CTMC is a tuple (S, s_1, P, ν, AP, L) where:

- S is a finite set of states;
- $s_1 \in S$ is the initial state;
- $P : S \times S \to [0, 1]$ is the transition probability matrix;
- ν is a vector of real values in $\mathbb{R}^{|S|}_{>0}$;
- AP is the set of atomic propositions;
- $L : S \to 2^{AP}$ is a labeling function that associates to each state the set of atomic propositions that are true in that state.

The CTMC evolves within the state space $S = \{s_1, \ldots, s_K\}$. When the process enters into state s_i, it remains there for an exponentially distributed time period t with mean $1/\nu_i$. At the end of t, the process will move to a different state s_j, with $j \neq i$ with probability $P(s_i, s_j)$, such that $\sum_j P(s_i, s_j) = 1$, and $\forall i, P(s_i, s_i) = 0$. In the rest of the paper, the notation p_{ij} will be used as short form for $P(s_i, s_j)$. The transition probability matrix is then denoted by $P = (p_{ij})$. It is worth noting that P defines an embedded DTMC model. In fact, by discarding the temporal aspects defined by ν, we obtain a discrete time stochastic processes with discrete state space, whose behavior is specified by P.

The vector $\nu = (\nu_1, \ldots, \nu_K)$ defines the parameters of K different exponential distributions governing the time spent in states $s_1, \ldots, s_K$, respectively. The parameter $r_{ij} = \nu_i \cdot p_{ij}$ yields the local rate, or intensity, of transitioning from state s_i to state s_j. The so computed rates define the infinitesimal generator matrix $R = (r_{ij})$ of the CTMC model.

3.2. Continuous Stochastic Logic

CSL (Baier et al., 2003) is a probabilistic branching-time temporal logic with state and path formulas interpreted over CTMC states and paths, respectively. It extends the Computation Tree Logic (Clarke et al., 2000) with two probabilistic operators that refer to the steady state and transient behavior of the target system. The steady-state operator $\mathcal{S}$ refers to the probability of residing in a particular set of states (identified by a state formula) in the long run. The transient operator $\mathcal{P}$ refers to the probability of the occurrence of particular paths in the CTMC. To predicate over the time required by a certain path, the path operators until (U) and next (X) are extended with a parameter that specifies a time interval. Formally, let $p \in [0, 1]$ be a real number, $\bowtie \in \{\leq, <, >, \geq\}$ a comparison operator, and $I \subseteq \mathbb{R}_{\geq 0}$ a non empty interval. The syntax of CSL state (ϕ) and path (ψ) formulas is defined inductively as follows.

$$\phi ::= true \mid a \in AP \mid \phi \wedge \phi \mid \neg\phi \mid \mathcal{S}_{\bowtie p}(\phi) \mid \mathcal{P}_{\bowtie p}(\psi) \quad (1)$$

$$\psi ::= \mathtt{X}\phi \mid \phi \ \mathtt{U} \ \phi \mid \phi \ \mathtt{U}^{I} \ \phi \quad (2)$$

Formally, the state-formulas are interpreted over the states of a CTMC with labels in AP. The definition of the entailment relation $\models$ is as follows. Let $Sat(\phi) = \{s \in S : s \models \phi\}$:

$$\begin{aligned} & s \models true \text{ for all } s \in S \\ & s \models a \text{ iff } a \in L(s) \\ & s \models \neg\phi \text{ iff } s \not\models \phi \\ & s \models \phi_1 \wedge \phi_2 \text{ iff } s \models \phi_i, i = 1, 2 \\ & s \models \mathcal{S}_{\bowtie p}(\phi) \text{ iff } \pi(s, Sat(\phi)) \in I_{\bowtie p} \\ & s \models \mathcal{P}_{\bowtie p}(\phi) \text{ iff } Prob(s, \phi) \in I_{\bowtie p} \end{aligned} \quad (3)$$

Here, $\pi(s, Sat(\phi))$ is defined as $\lim_{t\to\infty} Prob\{\sigma \in Path : \sigma_t \in Sat(\phi)\}$, where $Path$ is the set of all possible paths (i.e., sequences of states), and σ_t is the state of σ at time t. We use $Prob(s, \phi)$ to denote the probability measure of all paths $\sigma \in Path$ satisfying ϕ when the system starts in state s. Essentially, the formula $\mathcal{S}_{\bowtie p}(\phi)$ checks whether the steady-state probability for ϕ meets the boundary condition $\bowtie p$, whereas the state formula $\mathcal{P}_{\bowtie p}(\psi)$ asserts that the probability of observing the paths satisfying ψ meets $\bowtie p$. The usual CTL path quantifiers $\exists$, $\forall$ are replaced by the operator $\mathcal{P}$. Intuitively, $\exists\phi$ (i.e., there exists a path where ϕ holds) can be expressed as $\mathcal{P}_{>0}(\phi)$, $\forall\phi$ (i.e., for all paths ϕ holds) corresponds to $\mathcal{P}_{\geq 1}(\phi)$.

The entailment relation for the path-formulas is defined considering execution paths as follows, where we use $\sigma[k]$ to denote the kth state in the path σ:

$$\begin{aligned} & \sigma \models X\phi \quad \text{iff } \sigma[1] \text{ defined and } \sigma[1] \models \phi \\ & \sigma \models \phi_1 U \phi_2 \quad \text{iff } \exists k \geq 0 : (\sigma[k] \models \phi_2 \wedge \forall 0 \leq i \leq k, \sigma[i] \models \phi_1) \\ & \sigma \models \phi_1 U^I \phi_2 \quad \text{iff } \exists t \in I : (\sigma_t \models \phi_2 \wedge \forall u \in [0, u[, \sigma_u \models \phi_1) \end{aligned} \quad (4)$$

The operator U^I is the timed variant of the until operator U of CTL. The path formula $\phi_1 U^I \phi_2$ asserts that: ϕ_1 holds until some time instant $t \in I$; then ϕ_2 holds. Common derived path formulas are $F\phi$ (i.e., finally) and $G\phi$ (i.e., globally), defined as $trueU\phi$, and $\neg F \neg\phi$, respectively.

3.3. Bayesian inference

Bayesian inference (Berger, 1985) is a method of statistical inference used to update the probability for a hypothesis on uncertain (or unknown) quantities as more evidence or information becomes available. The main goal is to fill the gap between beliefs and evidence on one or more uncertain/unknown parameters θ affecting the behavior of a stochastic phenomenon of interest. The prior knowledge (i.e., initial hypothesis or belief) of θ is incrementally updated based on a collected data sample $y = (y_1, y_2, \ldots, y_n)$ describing the actual behavior of the target phenomenon. By using the Bayes' theorem we obtain the posterior distribution $f(\theta|y)$, describing the best knowledge of θ, given the evidence y.

$$f(\theta|y) \propto f(y|\theta) \cdot f(\theta) \quad (5)$$

The density $f(y|\theta)$ represents the compatibility of the data with the hypothesis. This density, when expressed as a function of θ, is usually referred to as the likelihood function and is generally denoted as $l(\theta|y)$, or $l(\theta|\text{data})$, when the notation gets heavy, following the notation introduced in Insua et al. (2012). The hypothesis is often available from external sources such as expert information based on past experience or previous studies. This information is encoded by the prior distribution $f(\theta)$. The posterior distribution can be used in turn to perform point and interval estimation. Point estimation is typically addressed in the multivariate case, by summarizing the distribution through the posterior mean $E[f(\theta|y)]$ and the (95%) Highest Density Region HDR$[f(\theta|y)]$. The magnitude of the HDR yields the highest possible accuracy in the estimation of the true value of θ and is usually adopted as a measure of the confidence gained after the inference process.

Table 1
Summary of valid requests for Sock Shop.

Service	ID	Path	Method
home	1	/index.html	GET
login	2	/login	GET
getCatalogue	3	/catalogue	GET
catalogueSize	4	/catalogue/size?size={}	GET
cataloguePage	5	/catalogue?page={}&size={}	GET
catalogue	6	/category.html	GET
getItem	7	/catalogue/{}	GET
getRelated	8	/catalogue?sort={}&size={}&tags={}	GET
showDetails	9	/detail.html?id={}	GET
tags	10	/tags	GET
getCart	11	/cart	GET
addToCart	12	/cart	POST
basket	13	/basket.html	GET
createOrder	14	/orders	POST
getOrders	15	/orders	GET
viewOrdersPage	16	/customer-orders.html	GET
getCustomer	17	/customers/{}	GET
getCard	18	/card	GET
getAddress	19	/address	GET

4. System under test

In this work, we use an existing microservices-based benchmark system called Sock Shop[3] as a running example and system under test in our empirical evaluation. Sock Shop is a microservice reference application used by researchers in the field of performance engineering to evaluate their approaches (e.g., see the studies Assunção et al., 2020; Grambow et al., 2020; v. Kistowski et al., 2019, to name a few). As described in Avritzer et al. (2020), Sock Shop supports: (*i*) usage of well-established microservice architectural patterns; (*ii*) possibility of using automated deployment practices in software containers; (*iii*) support for different deployment configuration options. Furthermore, it enables a direct quantitative comparison with existing performance analysis techniques adopting this system as a benchmark (Avritzer et al., 2018, 2020).

The system runs a containerized e-commerce web application composed of a number of microservices implemented by using various technologies, such as Java, .NET, Node.js, and Go. Table 1 summarizes the microservices and related requests that can be issued to them. Different types of users show different interaction patterns. For instance, a user that buys a product is likely to authenticate, add one or more product into the cart, and then create an order. This usage pattern reduces to a path of service invocations. For instance, surfing the catalog and adding products to the cart can be executed by means of the following path of valid requests: `cataloguePage`, `showDetails`, `getItem`, `getCustomer`, `getCart`, `addToCart`. Instead, a nominal *visitor* is likely to surf the catalog without authenticating and filling up the cart. In this latter case, the path of requests could be as follows: `cataloguePage`, `showDetails`, `getItem`,

[3] Sock Shop is an open source project maintained by Weaveworks. Sources and documentation are available at: https://microservices-demo.github.io/.

Table 2
Probabilistic timed reachability and response patterns.

Pattern	Natural language description	Formalization as CSL property
P_1	Probabilistic timed reachability: the state formula ϕ must become true within time bound t, with a probability bound $\bowtie p$.	$\mathcal{P}_{\bowtie p}\ [F^{<t}\phi\ ]$
P_2	Probabilistic response: after the state formula ϕ_1 holds, the state formula ϕ_2 must become true within time bound t, with a probability bound $\bowtie p$.	$\mathcal{P}_{\geq 1}\ [G\ (\ \phi_1 \rightarrow \mathcal{P}_{\bowtie p}\ [\ F^{<t}\phi_2\ ]\ )]$
P_3	Probabilistic constrained response: after the state formula ϕ_1 holds, the state formula ϕ_3 must become true, without ϕ_2 ever holding, within time bound t, with a probability bound $\bowtie p$.	$\mathcal{P}_{\geq 1}\ [G\ (\ \phi_1 \rightarrow \mathcal{P}_{\bowtie p}\ [\ (\ \neg\phi_2 U^{<t}\phi_3\ )\ ]\ )]$
P_4	Time constrained response chain (N causes, 1 effect): if ϕ_1 followed by $(\phi_i$ within time t_i, with a probability bound $\bowtie p_i)^{2\leq i\leq N}$ have occurred, then in response ϕ_{N+1} eventually holds within time bound t_{N+1}, with a probability bound $\bowtie p_{N+1}$.	$\mathcal{P}_{\geq 1}\ [G\ \phi_1 \rightarrow Ch(2)]$ with $Ch(i \leq N) = \mathcal{P}_{\bowtie p_i}[F^{<t_i}(\phi_i\ \&\ Ch(i+1))]$; and $Ch(i = N+1) = \mathcal{P}_{\geq 1}[X(\mathcal{P}_{\bowtie p_i}[F_{<t_i}\phi_i])]$

Table 3
System-level performance requirements for Sock Shop.

Requirement	Natural language description	Formalization as CSL property	Pattern
R_1	The visualization of the orders page in less than 1 min shall happen at least in 90% of the times.	$\mathcal{P}_{>0.9}\ [F^{<60}$ viewOrdersPage$]$	P_1
R_2	The catalog shall be reached in no more than 10 s after the login, at least 85% of the times.	$\mathcal{P}_{\geq 1}\ [G\ (\ $login$ \rightarrow$ $\mathcal{P}_{>0.85}\ [\ F^{<10}$getCatalogue$\ ]\ )]$	P_2
R_3	The payment of an order involves the getCard service, which is activated, in 98% of the cases, within 5 s from a getCart request being handled. Afterwards, a createOrder request is successfully executed within another 5 s in 98% of the cases.	$\mathcal{P}_{\geq 1}\ [G$ getCart $\rightarrow$ $(\mathcal{P}_{>0.98}\ [\ F_{<5}$ (getCard & $\mathcal{P}_{\geq 1}\ [\ X\ (\mathcal{P}_{\geq 0.98}\ [\ F_{<5}$ createOrder $])])])]$	P_4
R_4	The authentication shall last no more than 10 s and then reaching the orders page shall take no more than other 10 s in at least 65% of the times.	$\mathcal{P}_{>0.65}\ [F^{<10}$ login & $(\mathcal{P}_{>0}\ [F^{<10}$ viewOrdersPage $])]$	custom
R_5	In the long-run, high probability of a successful pay shall be achieved within 2 min, more than 90% of the times.	$\mathcal{P}_{<0.1}\ [\ F^{>120}\ S^{>0.9}[$ createOrder $]\ ]$	custom

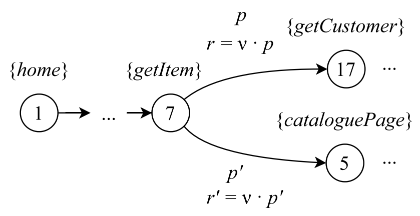


Fig. 2. Small extract of the CTMC model of Sock Shop.

`cataloguePage`, `showDetails`, `getItem`. The two user behaviors yield different request invocations (i.e., `getCustomer` and `cataloguePage`) after `getItem`. The CTMC extracted in Fig. 2 models these alternative paths with two outgoing edges from s_7 to s_{17} and s_5, respectively. In fact, our modeling approach maps requests to model states and possible sequences of requests to model paths. Model states are uniquely identified by the ID listed in Table 1, while the set of atomic propositions AP is defined by the set of request labels. The transition probabilities (e.g., p, p') depend on the distribution of the user categories. Considering our simple example, 70% visitors and 30% buyers reflect to $p = 0.7$ and $p' = 0.3$. In this sense, the transition probability matrix P (i.e., the embedded DTMC model) defines the usage profile, leveraged to guide the load testing activity as described in Section 5. Temporal aspects of the CTMC model (e.g., r, r') depend on the matrix P and the vector ν. While the former component P is (in our context) known, and given by a certain mix of user categories, the latter one ν is difficult or even impossible to fully specify at design-time, before observing the SUT at runtime.

In order to express requirements for the SUT we use CSL formulas, as discussed in Section 3. With the aid of this language, engineers can express properties about the system that are typically domain-dependent. To ease the formulation of probabilistic properties, engineers usually make use of property specification patterns (Autili et al., 2015). Patterns describe generalized and recurring properties of interest. In our case, we are interested in verifying how the target microservices system responds to user requests in terms of performance, therefore we restrict ourselves to properties that can be instanced by using probabilistic timed reachability and probabilistic response patterns (Grunske, 2008; Autili et al., 2015) adapted to the CSL syntax. Representative examples of patterns are reported in Table 2 showing a description in natural language as well as the corresponding formalization in CSL. In Table 3, we list a number of domain-dependent requirements that have been defined for Sock Shop. The table contains requirements instanced from the patterns as well as custom CSL properties. As an example, R_1, R_2, and R_3 are instances of P_1, P_2, and P_3, respectively, where state-formulas are microservices of Sock Shop and both probability bounds and time bounds are domain-dependent. In this case, the management of catalogs and orders are core functions from the perspective of the business logic of Sock Shop. Therefore, requirements predicate over the probabilistic response of the system when the microservices carrying out such functions are invoked.

Based on the assumptions reported above, such properties cannot be verified during the early design stages since the initial CTMC model is underspecified. Namely, the values of the infinitesimal generator matrix R are uncertain/unknown quantities that must be learned based on runtime evidence collected by executing the stages of our methodology described in the following section.

5. Test-based learning and verification

This section illustrates the steps of our methodology. The presentation follows the stages reported in Fig. 1. Namely, we introduce the preliminary stage in Section 5.1 (analysis of the operational setting) and then we describe the core stages in Section 5.2 (performance model learning) and Section 5.3 (verification).

Table 4
Examples of user categories interacting with Sock Shop.

User category	Weight	Description of the workflow	Corresponding sequence of requests	length
visitor	0.4	Visits the home page, views the catalog and the details of some products	home → getCatalogue → getCart → home → getCatalogue → getCart → catalog → catalogueSize → tags → cataloguePage → getCart → getCustomer → showDetails → getItem → getCustomer → getCart → getRelated	17
buyer	0.3	Visits the home page, logs in, views the catalog and some details, adds a product to the cart, visits the cart, and then creates an order	home → getCatalogue → getCart → login → home → getCatalogue → getCart → home → getCatalogue → getCart → catalog → catalogueSize → tags → cataloguePage → getCart → getCustomer → showDetails → getItem → getCustomer → getCart → getRelated → addToCart → showDetails → getItem → getCustomer → getCart → getRelated → basket → getCart → getCard → getAddress → getCatalogue → getItem → getCart → getCustomer → getItem → createOrder → viewOrdersPage → getOrders → getCart → getCustomer → getItem	42
order visitor	0.3	Visits the home page, logs in, and views the stored orders	home → getCatalogue → getCart → login → home → getCatalogue → getCart → viewOrdersPage → getOrders → getCart → getCustomer → getItem	12

5.1. Analysis of the operational setting

The objective of this preliminary phase is to provide the load testing with the ability to generate representative operational conditions to be tested. In our context, an operational condition is defined as a pair: *workload intensity* and *usage profile*. A detailed description of the two elements is as follows.

5.1.1. Workload intensity

The workload intensity (Kistowski et al., 2014) represents the number of (active) concurrent users in the SUT. The empirical distribution computed from operational data describes for each workload intensity $\lambda \in \Lambda$, its probability of occurrence, estimated as the relative occurrence frequency $f(\lambda)$ in the data gathered by observing the SUT. To build the empirical distribution, the number of concurrent users accessing the application shall be periodically recorded in a selected time window. From this data, we apply data binning to aggregate the recorded intensities into a smaller number of bins $\lambda_1, \ldots, \lambda_k$ that are then associated with the frequency of occurrence $f(\lambda_1), \ldots, f(\lambda_k)$. The test suite coverage criterion is based on the frequency values. The rationale is to use the empirical distribution to select the representative intensities to be generated during load testing. By relying on the operational data of our running example, we identified the aggregated bins reported in Eq. (6).

$$\begin{aligned} \Lambda &= \{50, 100, 150, 200, 250, 300\} \\ f(\lambda_i) &= \{0.11, 0.19, 0.21, 0.23, 0.20, 0.06\} \end{aligned} \tag{6}$$

5.1.2. Usage profile

In addition to intensity values, representative usage profiles must be generated to learn accurate performance models. A usage profile describes the categories of users (i.e., actors) that are likely to use the SUT in the target operational environment. We adopt a *behavior mix* modeling approach (Avritzer et al., 2020), where workload generation is conducted by using a weighted random sampling of users from a number of categories. Each category has a weight ω (i.e., selection probability) and specifies the behavior of a certain type of users through a sequence of valid requests to services exposed by the SUT. The categories listed in Table 4 (*visitor*, *buyer*, and *order visitor*) have been identified in Sock Shop to replicate the usage profile reported in Avritzer et al. (2020).

According to our framework in Fig. 1, both the categories and the behavior mix can be user-defined or extracted mechanically from operational data in the preliminary stage. The mechanical extraction adopted by our methodology follows the WESSBAS approach introduced in Vögele et al. (2018). In the following we briefly describe the required steps and we let the reader refer to this latter work for further details.

- *Session log generation*. While the system runs in production, raw session information can be recorded in a selected time window. For each user request, the following information shall be extracted: session identifier, request URL, method (e.g., GET, POST), request start time, request end time. Then, requests are grouped by the specified session identifier (e.g., session ID, or client IP address) to create an ordered sequence of service requests extracted from URLs and methods.
- *Clustering*. This step has the objective of identifying different groups of users with similar navigational patterns (i.e., user categories). As suggested in Vögele et al. (2018), the centroid-based X-means algorithm can be conveniently used in this case, since it scales better than K-means and it does not require the specification of the exact number of clusters in advance. Thus, each session log can be encoded into a matrix $C = (c_{ij})$ defining the transition counts for each pair of services (i, j). The encoded sessions are then clustered based on the normalized Euclidean distance metric. As a result, the centroids computed by the clustering algorithm represent the identified user categories.
- *Behavior mix extraction*. After the clustering, the relative frequency ω of each user category is calculated by dividing the number of sessions within each cluster by the total number of sessions in the session log.

When the categories have been defined, our framework automatically generates a formal representation of the behavior mix in terms of a DTMC model. The available requests in Table 1 are numbered to uniquely identify states in the DTMC (e.g., `home` is the initial state s_1). Given the user categories, for each state s_i we create the outgoing transitions in the DTMC by identifying all possible next requests considering all the user categories. More precisely, the transition probability p_{ij} is computed as follows:

$$p_{ij} = \sum_{c} P(s_i \rightarrow s_j | c) \cdot P(c) \tag{7}$$

with $P(c)$ equal to the relative frequency ω associated with the category c, and $P(s_i \rightarrow s_j | c)$ equal to the frequency of occurrence of $s_i \rightarrow s_j$ in the category c.

As an example, the transition probability $p_{3,7}$ (i.e., `getCatalogue` → `getItem`) is equal to $P($`getCatalogue` → `getItem` | *buyer*$) \cdot P($*buyer*$)$ since this transitions occurs only in the category *buyer*. Here, `getItem` follows `getCatalogue` in 1 out of 4 cases. Therefore, the transition probability $p_{3,7} = 0.25 \cdot 0.3 = 0.075$. By following the same approach, the transition probability $p_{3,11}$ (i.e., `getCatalogue` → `getCart`) is equal to $0.4 + 0.3 + 0.3 \cdot 0.75 = 0.925$.

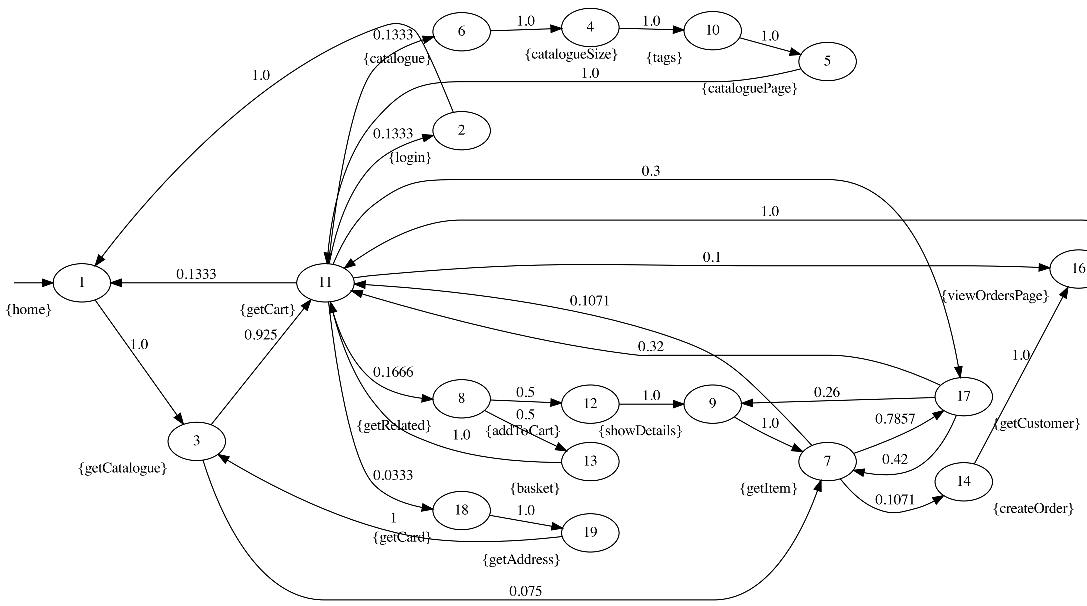


Fig. 3. DTMC model specifying the behavior mix of Sock Shop.

Fig. 3 shows the whole DTMC model mechanically obtained from the probability matrix $P = (p_{ij})$ computed by applying the process above. Such a model represents in our approach the behavior mix that drives the testing sessions in the learning stage, as described in Section 5.2.

It is worth noting that, even though each user category can be described in principle as DTMC following the approach presented in Vögele et al. (2018), in this work we follow the guideline reported in Avritzer et al. (2020) that characterizes a user category with a single workflow (sequence of requests). This choice is motivated by the need of carrying out a direct quantitative comparison between our approach and the Domain Metric (Avritzer et al., 2020), that we use as ground truth baseline as anticipated in Section 1. In case each category is described by a DTMC, the behavior mix can be computed by following Eq. (7) in a comparable way. The probability $P(s_i \to s_j|c)$ depends in this case on the transition probability matrix of the corresponding DTMC model rather than the frequency of occurrence in the sequence of requests.

5.2. Performance model learning

The objective of this phase is to learn performance models by running a number of load testing sessions to stress the SUT under different operational conditions and deployment configurations. The outcome of each testing session is a CTMC model that encodes temporal information to enable further analysis in the last stage (i.e., verification). In the following, we provide a description of the main theoretical aspects of the inference process and the termination condition as well as implementation details.

5.2.1. Inference process

Each testing session takes as input the DTMC model $\mathcal{M}$, a workload intensity λ, and a deployment configuration c. The *inference process* runs along with the testing session and it incrementally updates the prior knowledge taking into account the runtime evidence collected by testing the SUT deployed according to c. The testing session generates the workload λ where each synthetic user follows the stochastic behavior defined by $\mathcal{M}$. More precisely, each concurrent user decides the next request depending on the current state in $\mathcal{M}$ and the probability matrix (p_{ij}). For instance, from state s_3, the user performs either `getItem` or `getCart` with probability 0.075 and 0.925, respectively. The next state is either s_7 or s_{11}.

The goal of the inference process is to estimate the infinitesimal generator matrix $R = (r_{ij})$ of a CTMC model $\mathcal{X}$, given the priors and the runtime evidence. The evidence in our context is represented by the time spent in each state i and the occurrence of every transition from state i to state j as an event trace. It is worth noting that we observe λ concurrent users modeled through statistically independent instances of the DTMC $\mathcal{M}$, all starting from the common initial state. Namely, the transition matrix $P = (p_{ij})$ of $\mathcal{M}$ defines the embedded DTMC of the CTMC $\mathcal{X}$ to be inferred. The whole testing session executes the transition from state i to state j a number of times, denoted by N_{ij}. The total time spent in state i is denoted by T_i.

From the Bayesian statistics perspective, the problem reduces to infer the posterior probability $f(P, \nu|\text{data})$, where the transition matrix P is known, and the distributions of the time spent ν are uncertain or even unknown. Assume that we observe a sequence of states x_i and time instants t_i, for $i = 1, \dots, n$, of the first n transitions of the chain $\mathcal{X}$. Then, the likelihood function can be written as follows.

$$\begin{aligned} l(P, \nu|\text{data}) &= \prod_{i=1}^{n} \nu_{x_{i-1}} e^{-\nu_{x_{i-1}}(t_i - t_{i-1})} p_{x_{i-1}x_i} \\ &\propto \prod_{i=1}^{K} \nu_i^{N_i} e^{-\nu_i T_i} \prod_{i=1}^{K} p_{ij}^{N_{ij}} \end{aligned} \tag{8}$$

where $N_i = \sum_{i=1}^{K} N_{ij}$ is the total number of transitions out of state i, with $i, j \in \{1, \dots, K\}$, i.e., the finite state space of $\mathcal{X}$. The likelihood function in Eq. (8) can be written in the form $l(P, \nu|\text{data}) = l_1(P|\text{data})\, l_2(\nu|\text{data})$, where:

$$l_1(P|\text{data}) = \prod_{i=1}^{K}\prod_{j=1}^{K} p_{ij}^{N_{ij}}, \quad l_2(\nu|\text{data}) = \prod_{i=1}^{K} \nu_i^{N_i} e^{-\nu_i T_i} \tag{9}$$

This setting implies that, given independent priors for P and ν, the posterior distributions are independent and the inference process for ν (i.e., unknown parameters in our context) can be carried out apart from P (i.e., known parameters in our context). A natural conjugate prior for the transition rates ν is the Gamma distribution (Diaconis et al., 1979). Given the following independent gamma prior distributions,

$$f(\nu_i) \sim Ga(a_i, b_i), i = 1, \dots, K \tag{10}$$

the posterior distribution (combining prior and likelihood as described in Section 3) is defined as follows:

$$f(\nu_i|\text{data}) \sim Ga(a_i + N_i, b_i + T_i) \tag{11}$$

Eq. (11) is the so-called updating rule used at runtime while collecting the evidence. It is worth noting that the updating rule, used to produce the new estimates, is computationally inexpensive. For this reason, we can apply it online along with testing, whereas the verification process, which is potentially more expensive, is kept separate as the last (offline) stage.

In case the tester does not have initial beliefs on transition rates, it is quite common to use an uninformative prior, such as $f(\nu_i) \sim Ga(0, 0)$ for all i. Even though this distribution is *improper* (Robert, 2007), we obtain a sensible posterior as soon as we get empirical observations. Nonetheless, the tester can build the priors to reflect initial beliefs based, for instance, on past experience or previous studies, as described in Robert (2007). In this case, the tester can choose suitable parameters: *shape* a_i and *rate* b_i. As an example, consider the `getItem` request to the catalog microservice (i.e., state s_7). If the expectation for the `getItem` permanence rate is 3.0, we could adopt either $f(\nu_7) \sim Ga(6, 2)$ or $f(\nu_7) \sim Ga(12, 4)$. Although the two distributions have the same mean value 3.0, the degree of confidence measured through the HDR is different. More precisely, the information expressed by the latter example is stronger: the HDR in the former case is larger [0.84, 5.39], while in the latter case is smaller [1.43, 4.76], meaning higher confidence.

When a testing session meets the termination condition, the inference module stops and then applies *punctual summarization*. Namely, we update the parameters of the infinitesimal generator matrix $R = (r_{ij})$ of the model $\mathcal{X}$ by applying $r_{ij} = \bar{\nu}_i\, p_{ij}$, where $\bar{\nu}_i$ is estimated by summarizing the posterior knowledge through $E(f(\nu_i|\text{data}))$, as described in Section 3.

5.2.2. Termination condition

The termination condition of each testing session is determined by a statistic-driven technique (Jiang and Hassan, 2015) that aims at ensuring the accuracy of the model learned from collected data. This approach allows us to overcome shortcomings of pre-defined static configurations (e.g., timer-driven and counter-driven) by stopping a testing session as soon as the posterior knowledge converges to statistically stable model parameters. Specifically, our approach adopts Bayesian model comparison to detect when the difference between two alternative models is not substantial. Given a sample size N, the inference process builds a sequence of posterior models as follows:

$$M_1 = f(\nu|\text{data}_1), \dots, M_n = f(\nu|\text{data}_n) \tag{12}$$

where M_{i+1} is the model learned by accounting for the prior M_i and the collected evidence data_{i+1} having sample size N. Thus, for each $i > 1$, we apply a model selection criterion based on the *Bayes factor* (Jeffreys, 1998), reported below:

$$\mathcal{K}(M, M') = \frac{P(\text{data}|M)}{P(\text{data}|M')} \tag{13}$$

Essentially, Eq. (13) quantifies the support for the model M over the model M' given the data under consideration. As described in Jeffreys (1998), the standard scale for interpretation of this ratio states that a $\mathcal{K}$ value between 10^0 and $10^{1/2}$ is barely worth mentioning. Thus, we exploit the ratio $\mathcal{K}$ to terminate a testing session as soon as we find the iteration j, such that $\mathcal{K}(M_j, M_{j-1})$ falls in this interval.

5.2.3. Implementation

Fig. 4 illustrates the main components of our implementation supporting the model learning stage of the whole methodology. The schema contains the *testing infrastructure* (composed of *Load driver node* and *SUT node*) and the two main modules. The *load testing module* orchestrates the testing sessions. The *inference module* executes the inference process and checks the termination condition.

Testing infrastructure. The testing infrastructure defines the in-vitro environment where the SUT is deployed and then tested. Our approach aims at reproducing the conditions that are likely to happen in production using an in-vitro setting. The testing tool synthetically generates the workload based on the given DTMC model and the empirical distribution of the workload intensity. Our testing infrastructure supports standard practices used in the context of microservices applications, i.e., containerized deployment upon either bare metal or virtualized computing nodes. Fig. 4 shows that there exist two main computing nodes in a typical leader–follower configuration: the Load driver node (i.e., the leader) and the SUT node (i.e., the follower). We rely on Docker for the deployment of the containerized application. Specifically, the Docker Swarm mechanism allows the Docker manager component to take control over the Docker worker by means of deploy/undeploy services with fine-grained configuration settings for the Docker containers. The whole SUT consists of a number of Docker containers. Each container executes a single microservice s_i of the SUT.

Load testing module. As shown in Fig. 4, the core component of this module has been realized using the Locust[4] load testing tool. In our toolchain, Locust handles the creation and execution of the workload driven by the DTMC specification. On top of it, we created an orchestration layer that automates the deployment/undeployment of the SUT and the execution/monitoring of multiple testing sessions. The orchestration layer relies on the BenchFlow domain specific language (Ferme and Pautasso, 2018) to declare the testing sessions. The language allows the following input parameters to be defined:

- the DTMC model $\mathcal{M}$ that drives the synthetic users;
- the empirical distribution $f(\lambda_1), \dots, f(\lambda_k)$ for each workload intensity in Λ;
- the set $\mathcal{C}$ of deployment configurations, each one defining the amount of RAM, CPU share, replicas per microservice.

For each element $\langle\lambda, c\rangle \in \Lambda \times \mathcal{C}$, the orchestrator creates and executes a testing session. Docker deploys the SUT using configuration c. Then Locust generates the workload intensity λ, where each synthetic user follows the stochastic behavior defined by $\mathcal{M}$, as described in Section 5.2.1. During the testing session, Locust reports all the issued requests and their response time. This information represents the run-time evidence that feeds the inference module to compute the posterior density functions.

Inference module. The inference module runs in parallel with the load testing module and is fed by Locust that periodically samples the occurrences N_i of each state i and the total amount of time T_i spent in state i. This information is used to build the posterior knowledge by applying the Bayesian inference process, as detailed in Section 5.2.1. For each collected sample, the module produces a new set of posterior density functions that are compared to those computed with the previous sample using the Bayes factor, as described in Section 5.2.2. Such a process continues until, according to the Bayes factor, the posteriors are

[4] Locust is open source software available at http://locust.io.

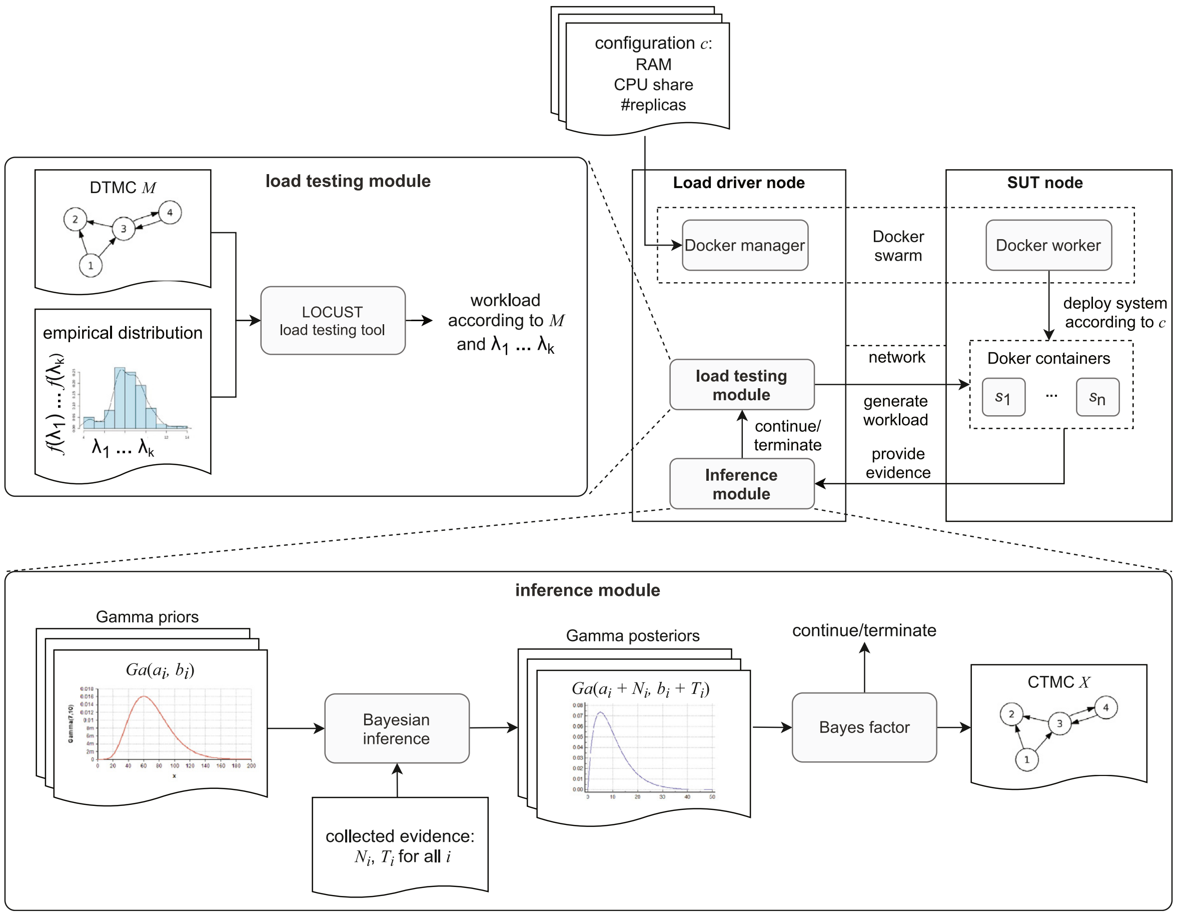


Fig. 4. Schema of the testing infrastructure and zoom in into the load testing module and the inference module.

not statistically stable which represents our termination condition. As soon as the Bayes factor converges, the inference module triggers a termination signal issued to the load testing module. The outcome of a testing session, executed for the pair $\langle \lambda, c \rangle$, is a CTMC model that composes the input of the *verification* stage of our methodology.

5.3. Verification

The objective of this last phase is to verify the learned model against formal requirements. Given a CTMC model $\mathcal{X}_i$, learned in each testing session $i = 1, \dots, n$, we use the PRISM probabilistic model checker to detect possible violations of the CSL properties and compute the configuration score. The verification stage follows the two steps reported below.

5.3.1. Verification of system requirements

This step verifies the domain-dependent requirements of interest crafted by engineers as CSL properties upon each individual model $\mathcal{X}_i$. The outcome of this step is a report containing, for each model, the list of satisfied/unsatisfied requirements to recognize problematic pairs $\langle \lambda, c \rangle$ of deployment configurations and workload intensities. As an example, the CSL properties reported in Table 3 represent relevant system-level performance requirements defined for the Sock Shop case study. These properties have been verified in our empirical evaluation to assess the applicability of the approach and the effectiveness in detecting problematic deployment configurations that might break requirements. As an example, Table 5 lists the outcome we obtained by verifying R_1–R_5 upon the CTMC models learned from different testing sessions executed by varying the deployment configuration and the workload intensity. In this example, we keep fixed the amount of memory and CPU share (8 GB and 25%, respectively) and we change the number of replicas of the `cart` microservice (2 and 4 replicas). Two testing sessions per configuration have been executed by generating a workload intensity of 50 and 300, respectively (i.e., the lowest and the highest intensity values of the discretized empirical distribution). As shown in Table 5, the verification outcome is positive (5 out of 5 satisfied requirements) for the configurations (a) and (b) when the load intensity is lower (150). With a higher load intensity (300), the outcome is negative (3 out of five unsatisfied requirements) in case we adopt (b) 4 `cart` replicas rather than (a) 2 `cart` replicas.

5.3.2. Configuration score

This step is used to assess the performance of the SUT as a whole by evaluating the scalability attitude of each deployment configuration. Firstly, we generate and verify a number of CSL properties to measure the performance of each microservice of the target system. These properties are automatically generated starting from the template in Eq. (14) that represents a probabilistic constrained response meta-property defined for all service s of the SUT (e.g., the services reported in Table 1 for Sock Shop).

$$\mathcal{Z}_s(T) = \mathcal{P}_{\geq 1}[G(s \rightarrow \mathcal{P}_{=?}[(sU^{\leq T} \neg s)])] \tag{14}$$

The property $\mathcal{Z}_s(T)$ quantifies the probability that the response time of the service s is lower than a given parametric threshold T. Thus, for each service, we instantiate the property and we use the model checker to compute the probability by varying the

Table 5
Verification of system-level requirements R_1–R_5 with two target configurations by varying the workload.

Deployment configuration	Workload intensity	R_1	R_2	R_3	R_4	R_5
(a) 8 GB, 25% CPU share, 2 × `cart` replicas	50	✓	✓	✓	✓	✓
(b) 8 GB, 25% CPU share, 4 × `cart` replicas	150	✓	✓	✓	✓	✓
(a) 8 GB, 25% CPU share, 2 × `cart` replicas	300	✓	✓	✓	✓	✓
(b) 8 GB, 25% CPU share, 4 × `cart` replicas	150	✗	✗	✓	✗	✓

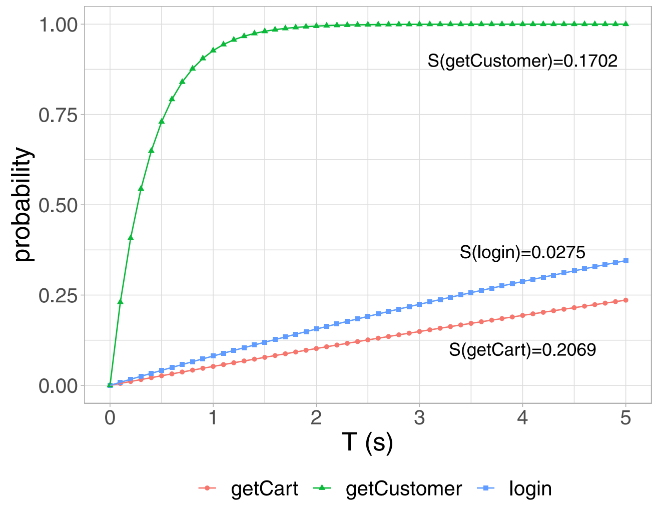


Fig. 5. Probability curves computed by instantiating $\mathcal{Z}_s(T)$.

parameter T. The objective is to conduct a quantitative analysis of the likelihood of observing a response time up to T, with T parametric threshold varying between a lower- and upper-bound lb and ub, respectively. Even though each curve $\mathcal{Z}_s(T)$ depends on how the microservice s performs rather than the entire CTMC model, we adopt the model checker to operationalize the computation of the configuration score and adopt a unified approach in the whole verification stage. The model checker is used to automatically produce the cumulative distribution functions of the response time inferred during the testing sessions, as shown in Fig. 5. This example illustrates the curves $\mathcal{Z}_s(T)$ computed by testing Sock Shop under the configuration (b) in Table 5, applying a workload intensity of 300. The three curves refer to three different microservices: `getCart`, `getCustomer`, and `login`, under parametric variability of T in $[0, 5]$ s. The performance of the microservice `getCustomer` is higher. We can observe that the probability of observing a response time less than 2 s is close to one. The other two microservices (i.e., `getCart`, and `login`) yield instead worse performance compared to the former one. The growth is slower and for both the microservices, the probability of observing a response time less than 2 s is less than 0.25.

The outcome described above is used as an aid to rank deployment configurations by leveraging the notion of *configuration score*. Such a score represents the evaluation of the system as a whole conducted by analyzing the performance models as defined by Eq. (14). Namely, the score measures the area under the curve $\mathcal{Z}_s(T)$ for all s, with workload intensity λ, as follows:

$$\varphi_\lambda(lb, ub) = \sum_s \int_{lb}^{ub} \mathcal{Z}_s(T)\,\mathrm{d}T \cdot \mathcal{S}(s) \tag{15}$$

The best deployment configuration yields the CTMC model $\mathcal{X}_i$ that maximizes the score $\varphi_\lambda(lb, ub)$. Intuitively, for each service, the bigger the area, the better the performance. Furthermore, the contribution of each individual service s to the performance of the system as a whole depends on the frequency of accessing s, estimated by computing the steady-state probability $\mathcal{S}(s)$ from the embedded DTMC model. For instance, according to Fig. 5, the steady-state probability $\mathcal{S}(\texttt{getCart}) = 0.2069$ is higher than $\mathcal{S}(\texttt{getCustomer}) = 0.1702$. Hence, poor performance associated with `getCart` has higher impact (i.e., higher weight) on the score φ_λ compared to the high performance of `getCustomer`.

Once the configuration score has been computed for each workload intensity λ, we summarize the *total score* of a configuration by applying a weighted sum over the λ values, leveraging on the frequency of occurrence $f(\lambda)$:

$$\Phi(\Lambda, lb, ub) = \sum_{\lambda \in \Lambda} \varphi_\lambda(lb, ub) \cdot f(\lambda) \tag{16}$$

As an example, according to the empirical distribution in Eq. (6) of our case study, the configuration score associated with $\lambda = 200$ is the most important (i.e., highest frequency equal to 0.23) whereas the score associated with $\lambda = 300$ is the less important (i.e., lowest frequency equal to 0.06).

6. Evaluation

We introduce the research questions in Section 6.1 that guided the evaluation of our approach. We describe the design of the evaluation in Section 6.2 and then we present the major results in Section 6.3. We then discuss threats to validity in Section 6.4.

6.1. Research questions

The major goal of the evaluation is to investigate the benefits and the costs of our approach supported by a software toolchain.[5] Starting from this main objective, we aim at answering the following research questions:

RQ1: What is the effectiveness in detecting violations of requirements?
RQ2: What is the effectiveness of the configuration score in ranking deployment alternatives?
RQ3: What is the cost of the model learning process?
RQ4: What is the cost of the verification process?

6.2. Design of the evaluation

To answer the four research questions, we designed a set of controlled experiments using Sock Shop as SUT (see Section 4). The machines used in our in-vitro testing infrastructure have the following characteristics: load driver node equipped with 4 GB RAM, 1 core at 2600 MHz; and SUT node equipped with 16 GB RAM, 4 cores at 2600 MHz. Both machines use magnetic disks with 15000 rpm and are connected using a shared 10 Gbit/s network. In our experiments, we adopted the empirical distribution of workload intensity Λ introduced in Eq. (6) and the usage profile described in Section 5.1.2. Assuming this operational setting, we executed a number of testing sessions varying the factors composing the deployment configuration as described in Section 5.2: the amount of RAM, CPU share, and service replicas. Specifically, for each factor, we considered the following values according to the resource constraints imposed by our in-vitro testing environment:

[5] The implementation of the core stages of our methodology is publicly available at https://github.com/pptam/pptam-tool. The replication package of experiments is available at https://doi.org/10.5281/zenodo.5078110.

Table 6
Deployment configurations.

ID	Memory (GB)	CPU (% share)	#replicas
1	8	50	1
2	8	50	2
3	8	50	4
4	8	25	1
5	8	25	2
6	8	25	4
7	16	50	1
8	16	50	2
9	16	50	4
10	16	25	1
11	16	25	2
12	16	25	4

- amount of RAM in [8 GB, 16 GB];
- CPU share in [25%, 50%];
- number of `cart` replicas in [1, 2, 4].

The latter point, in particular, is justified by the high volume of requests directed to the microservice `cart` during the testing sessions. Namely, according to the behavior mix represented by the DTMC model of Sock Shop, the steady-state probability $\mathcal{S}(\texttt{getCart})$ is the highest one. Therefore, the number of concurrent requests directed to this service during the testing sessions is higher compared to the other services.

The testing sessions executed during the experimental campaign were driven by: the DTMC model reported in Fig. 3; 6 aggregated workload intensity bins $\Lambda = \{50, \ldots, 300\}$ reported in Eq. (6); and 12 deployment configurations $\mathcal{C}$ listed in Table 6 derived from the combination of the factors listed above. For each one of the 72 settings in $\Lambda \times \mathcal{C}$, we executed each testing session[6] multiple times as further discussed below. For each session, we executed the model learning and then the verification steps (i.e., the two core stages of our methodology). The model learning has been executed along with the testing onto the in-vitro testing environment introduced above, whereas the verification stage has been executed on commodity hardware, i.e., a laptop equipped with a 2300 MHz dual-core CPU and 8 GB of RAM. We measured the effectiveness in terms of verification of performance requirements as well as the ability to rank deployment configurations based on the configuration score. The cost has been measured by taking into account the number of tests and the resources (execution time and memory consumption) required by the learning process and the verification one. Results have been compared to a ground truth baseline constructed by applying a selected state-of-the-art scalability assessment approach called Domain Metric (DM) (Avritzer et al., 2020), as anticipated in Section 1. This choice is justified by the fact that the DM has been shown effective in ranking configurations based on the ability to fit the operational setting. Its applicability has been recently demonstrated in different industrial domains (Avritzer et al., 2021b). Furthermore, it has been applied to the Sock Shop benchmark by using an in-vitro testing environment that we were able to replicate during our experiments, thus enabling a direct quantitative comparison of results.

6.3. Results

Here we discuss the most relevant results and we refer the reader to our implementation and dataset for the replicability of the experiments.

[6] In each testing session we adopted a *think time* modeled as a negative exponential distribution (with 0 s, 1 s, and 5 s for minimum, mean, and maximum think time, respectively, and an allowed deviation of 5% from the defined think time) to represent realistic user behavior as reported in Avritzer et al. (2020).

6.3.1. Results for RQ1

What is the effectiveness in detecting violations of requirements?

To address the first research question we conducted a set of controlled experiments to assess the verification stage of our approach. Here we focus on the ability of spotting violations of CSL requirements after the learning stage. Such assessment has been conducted by following two steps: (i) definition of a ground truth baseline; and then (ii) comparison of the verification outcome and the baseline under alternative learned models by varying the total cost of the learning stage (in terms of number of executed tests). The ground truth baseline is composed of a set of CSL requirements[7] having known outcome (i.e., either true or false) for each deployment configuration. More precisely, for each one of the 72 settings in $\Lambda \times \mathcal{C}$, we defined 20 system-level requirements: 10 true properties and 10 false properties. The value K represents the bounded cost (number of tests) necessary to achieve our termination condition based on the Bayes factor. Thus, we observed how the verification outcome changes when limiting the cost up to K. Specifically, to assess the ability to detect violations, we verified the baseline properties on the CTMC models built by limiting the cost as follows: $K - 50\%$, $K - 25\%$, $K - 12\%$, and K, respectively.

For each run, we measured the number of verification errors in the following two categories: E (i.e., true outcome for a property that shall not hold), and E' (i.e., false outcome for a property that shall hold). For each deployment configuration (1 to 12) and each workload intensity (50 to 300), we applied the learning process by varying the cost ($K - 50\%$ to K). Then, for each model, we verified the 20 baseline properties to compare the verification outcome with the oracle models. By following this process, we verified in total 7200 CSL properties for which we recorded E and E' occurrences. Fig. 6 shows the major results. For each deployment configuration, a bar plot shows the rate of E and E' errors. We can observe the highest error rate occurs with the lowest number of tests $K - 50\%$ (i.e., 0.74 and 0.73 on average for E and E', respectively). By increasing the total cost, both E and E' decrease down to zero when we set the cost equal to K. With $K - 12\%$ tests, we can observe a low error rate (i.e., 0.1 and 0.09 on average for E and E', respectively). With this setting, zero errors occurred in certain cases: 7 out of 12 times in both E and E'. According to the results in Fig. 6, K represents the safest choice. In this case, the error rate is zero for all the deployment configurations.

After the assessment described above, we verified the system-level requirements R_1–R_5 in Table 3 upon all models learned with total cost K to detect violated requirements when varying the workload intensity for each available deployment configuration of Sock Shop. The results of these experiments are reported in Table 7. We can observe that up to 100 concurrent users, all the configurations meet the performance requirements. From the workload intensity 150 to 300, we can observe instead a number of violations. In particular, we found that 6 out of 12 configurations do satisfy all the requirements, and increasing the number of replicas does not always guarantee better performance. According to our results, the system supports more than 2 `cart` replicas only when the CPU share is at least 50%.

RQ1 summary: the termination condition based on the Bayes factor represents an effective method. The results show that K tests are enough to learn the performance models with high accuracy. Further reduction of the total cost is likely to cause verification errors. The verification stage can be used by

[7] The CSL requirements composing the ground truth baseline are available in the replication package of the evaluation.

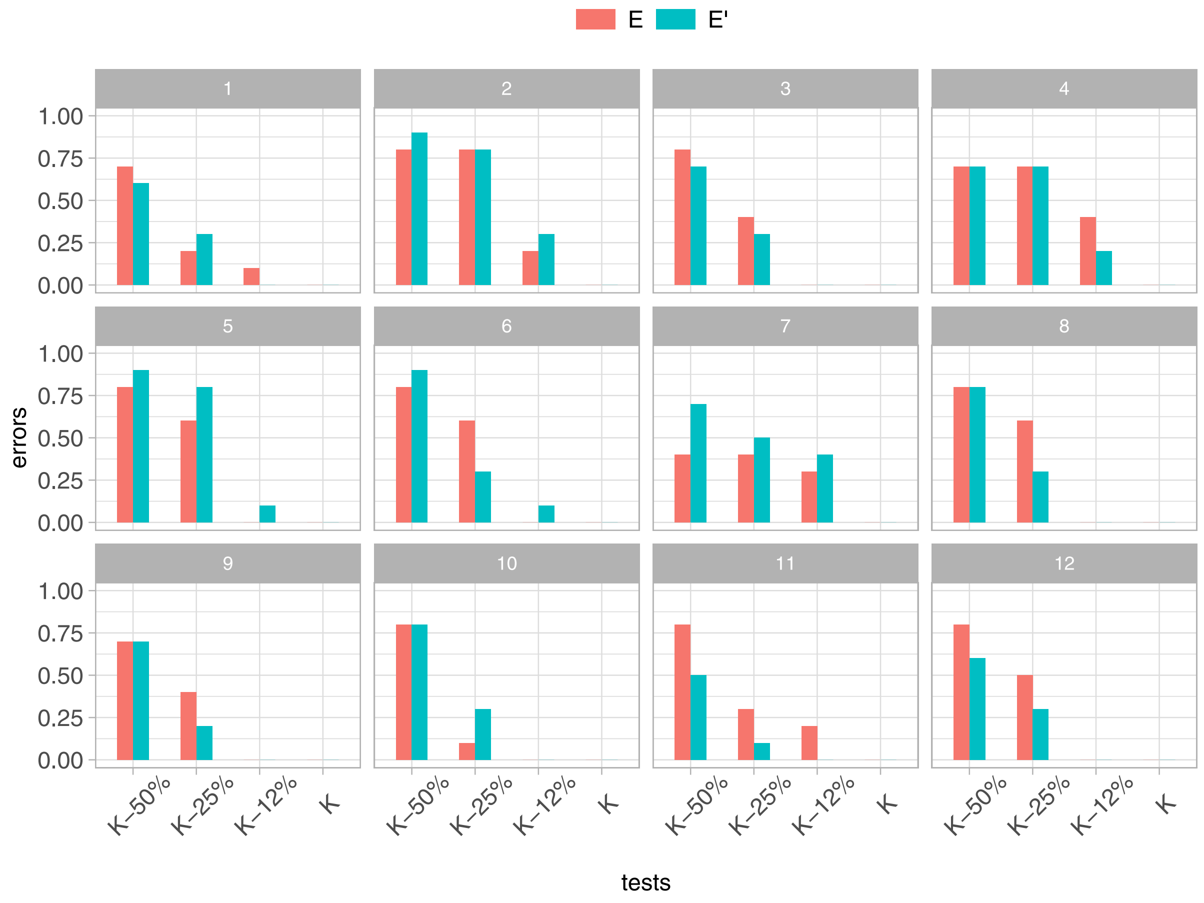
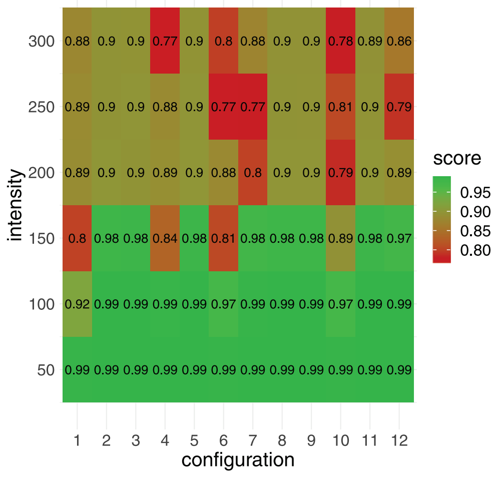
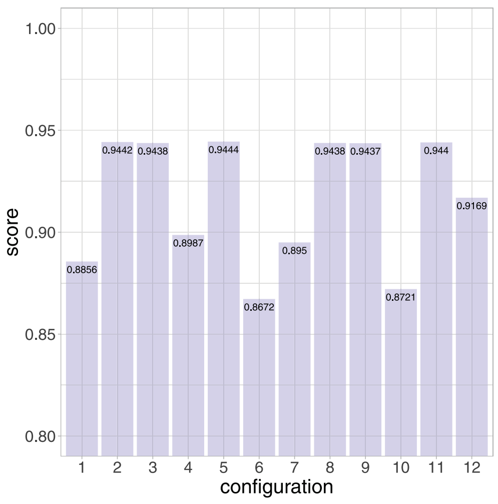


Fig. 6. Rate of verification errors depending on the number of tests up to K (i.e., the limit given by the Bayes factor).



intensity	1	2	3	4	5	6	7	8	9	10	11	12
300	0.88	0.9	0.9	0.77	0.9	0.8	0.88	0.9	0.9	0.78	0.89	0.86
250	0.89	0.9	0.9	0.88	0.9	0.77	0.77	0.9	0.9	0.81	0.9	0.79
200	0.89	0.9	0.9	0.89	0.9	0.88	0.8	0.9	0.9	0.79	0.9	0.89
150	0.8	0.98	0.98	0.84	0.98	0.81	0.98	0.98	0.98	0.89	0.98	0.97
100	0.92	0.99	0.99	0.99	0.99	0.97	0.99	0.99	0.99	0.97	0.99	0.99
50	0.99	0.99	0.99	0.99	0.99	0.99	0.99	0.99	0.99	0.99	0.99	0.99

(a) Configuration score $\varphi_{\lambda}(0, 5)$ per workload intensity λ.



(b) Total score $\Phi(\Lambda, 0, 5)$ per configuration.

Fig. 7. Scores computed considering the 12 target configurations.

engineers to understand performance issues and trace them back to requirements. Furthermore, automated verification can be used as an aid to prevent problematic configurations from being adopted in production. We were able to spot 6 out of 12 deployment configurations that do not meet all the performance requirements.

6.3.2. Results for RQ2

What is the effectiveness of the configuration score in ranking deployment alternatives?

This research question aims at investigating the ability of our methodology to detect both problematic deployment configurations based on the *configuration score* introduced in Section 5.3. We addressed this question by computing the individual score

Table 7
Verification of system-level requirements per configuration and workload intensity.

Configuration	Workload intensity	R_1	R_2	R_3	R_4	R_5	Configuration	Workload intensity	R_1	R_2	R_3	R_4	R_5
1	50	✓	✓	✓	✓	✓	2	50	✓	✓	✓	✓	✓
	100	✓	✓	✓	✓	✓		100	✓	✓	✓	✓	✓
	150	✗	✗	✓	✗	✓		150	✓	✓	✓	✓	✓
	200	✗	✓	✓	✓	✓		200	✓	✓	✓	✓	✓
	250	✗	✓	✓	✓	✓		250	✓	✓	✓	✓	✓
	300	✗	✗	✓	✓	✓		300	✓	✓	✓	✓	✓
3	50	✓	✓	✓	✓	✓	4	50	✓	✓	✓	✓	✓
	100	✓	✓	✓	✓	✓		100	✓	✓	✓	✓	✓
	150	✓	✓	✓	✓	✓		150	✗	✗	✓	✗	✓
	200	✓	✓	✓	✓	✓		200	✗	✓	✓	✓	✓
	250	✓	✓	✓	✓	✓		250	✗	✗	✓	✓	✓
	300	✓	✓	✓	✓	✓		300	✗	✗	✓	✗	✓
5	50	✓	✓	✓	✓	✓	6	50	✓	✓	✓	✓	✓
	100	✓	✓	✓	✓	✓		100	✓	✓	✓	✓	✓
	150	✓	✓	✓	✓	✓		150	✗	✗	✓	✗	✓
	200	✓	✓	✓	✓	✓		200	✗	✗	✓	✓	✓
	250	✓	✓	✓	✓	✓		250	✗	✗	✓	✗	✓
	300	✓	✓	✓	✓	✓		300	✗	✗	✓	✗	✓
7	50	✓	✓	✓	✓	✓	8	50	✓	✓	✓	✓	✓
	100	✓	✓	✓	✓	✓		100	✓	✓	✓	✓	✓
	150	✓	✓	✓	✓	✓		150	✓	✓	✓	✓	✓
	200	✗	✗	✓	✗	✓		200	✓	✓	✓	✓	✓
	250	✗	✗	✓	✗	✓		250	✓	✓	✓	✓	✓
	300	✗	✗	✓	✓	✓		300	✓	✓	✓	✓	✓
9	50	✓	✓	✓	✓	✓	10	50	✓	✓	✓	✓	✓
	100	✓	✓	✓	✓	✓		100	✓	✓	✓	✓	✓
	150	✓	✓	✓	✓	✓		150	✗	✗	✓	✓	✓
	200	✓	✓	✓	✓	✓		200	✗	✗	✓	✗	✓
	250	✓	✓	✓	✓	✓		250	✗	✗	✓	✗	✓
	300	✓	✓	✓	✓	✓		300	✗	✗	✓	✗	✓
11	50	✓	✓	✓	✓	✓	12	50	✓	✓	✓	✓	✓
	100	✓	✓	✓	✓	✓		100	✓	✓	✓	✓	✓
	150	✓	✓	✓	✓	✓		150	✓	✓	✓	✓	✓
	200	✓	✓	✓	✓	✓		200	✗	✓	✓	✓	✓
	250	✓	✓	✓	✓	✓		250	✗	✗	✓	✗	✓
	300	✓	✓	✓	✓	✓		300	✗	✗	✓	✓	✓

φ_λ for each testing session in $\Lambda \times \mathcal{C}$ as well as the total one Φ for each configuration in $\mathcal{C}$. For each session, we instanced the meta-property $\mathcal{Z}_s(T)$ reported in Eq. (15) for all the microservices of Sock Shop by varying the parametric threshold T from 0 to 5 with a step of 0.1 s. Thus, we automatically generated 50 CSL properties for each one of the 19 microservices. In total, 5700 properties have been generated to compute φ_λ for all $\lambda \in \Lambda$, and 68400 properties have been generated to compute Φ for all configurations. To assess whether the configuration score represents a suitable metric we checked whether the results are consistent with the verification of system-level requirements in Table 7. Furthermore, we compared them to a ground truth baseline constructed by applying the DM scalability assessment.

Fig. 7(a) shows the outcome of the computation of $\varphi_\lambda(0, 5)$, whereas Fig. 7 shows the total score $\Phi(\Lambda, 0, 5)$. Consider the heat-map in Fig. 7(a). The configuration score is high for all the configurations under a workload intensity up to 100 (average score equal to 0.98). From 150 to 300 users we observe a degradation of the performance denoted by a lower configuration score (average score equal to 0.88). Some of the configurations are likely to exhibit lower scores. As an example, the scores observed with configuration 5 are higher than or equal to 0.9, whereas configuration 6 yields scores lower than 0.9 in 4 out of 6 workload intensity bins. Such a difference is visible by observing the total score shown in Fig. 7 through a bar plot. Here we can see a substantial gap between configurations 5 and 6. The former one is likely to better fit the target operational setting since the total score is almost 10% higher. The total score reported in Fig. 7 is (mostly) consistent with the verification outcome reported in Table 7. The six configurations that meet all the system-level requirements are R_1–R_5. These configurations map to the highest total score ($\sim$0.94). The lowest score has been assigned to configuration 6 (i.e., 8 GB memory, 25% CPU share, 4 replicas). Such a result is comparable to what has been shown through the DM approach reported in Avritzer et al. (2020). In particular, the top scores we found in our experiments are the configurations that better fit the operational setting according to the DM.

It is worth noting that the total score reported in Fig. 7 yields aggregate information that cannot reasonably be used to identify the punctual CSL requirements that hold or do not hold. Indeed, requirements might predicate on microservices whose weight is low according to the usage profile. In this case, low performance exhibited by these services may have little impact on the total score even though they invalidate requirements. For instance, according to Fig. 7(a), configurations 1 and 4 exhibit a comparable score under workload intensity 250 (0.89 and 0.88, respectively), but configuration 4 does not meet R_2 which instead holds with configuration 1 (see Table 7). Furthermore, under workload intensity 300, configuration 4 yields a lower score compared to configuration 1 (0.77 and 0.88, respectively). However, this has a small impact on the total score since, according to Eq. (6), the frequency associated with intensity 300 is the lowest one (0.06). In Fig. 7, we can observe that even if the total score of configuration 4 is higher with respect to configuration 1, the latter one satisfies a higher number of requirements across the workload intensities, as reported in Table 7.

Based on our results and the related discussion reported above, the computation of the total score cannot substitute the verification of system-level requirements but provides engineers with complementary information. Specifically, the score can be used to extract insights on scalability issues depending on the configuration parameters. According to Fig. 7, we can observe that horizontal scaling does not guarantee better performance. With the resource constraints imposed by our in-vitro testing infrastructure, resources allocated to vertical scaling are not enough to support more than 2 microservice replicas when less than 50% CPU share is adopted.

RQ2 summary: we observed that a lower total score is likely assumed to be a bad smell of worse performance. Nevertheless, since the score holds aggregate information it is not enough to determine to what extent system-level requirements are satisfied. Therefore, the computation of the score cannot substitute the verification of requirements, but it provides engineers with complementary insights. Indeed, the score can guide engineers during the final "go live" decision gate in the selection of the deployment configuration. According to our results, we spot problematic combinations of configuration parameters that are likely to reduce the score. We found that vertical resources are not enough to support more than 2 microservice replicas when less than 50% CPU share is adopted.

6.3.3. Results for RQ3

What is the cost of the model learning process?

This research question aims at quantifying the total cost required by the testing sessions in terms of total number of tests (i.e., number of requests issued to microservices) as well as the execution time needed to achieve termination. To address this question, we conducted a set of experiments by taking control over the prior knowledge which represents the factor having major impact on the Bayesian inference carried out during the learning stage. Namely, for each deployment configuration, we executed a number of testing sessions by varying the HDR magnitude of the prior knowledge. As introduced in Section 3, the HDR magnitude represents in Bayesian inference the degree of confidence. The smaller the magnitude, the higher the confidence. Thus, we constructed 5 different priors where we embedded incremental knowledge to reduce the HDR magnitude. To build the priors we first executed the inference process using an uninformative one (i.e., HDR reduction equal to 0%). During the inference, we computed the HDR of the posterior density function. Thus, we extracted the posteriors having magnitude reductions of 20%, 40%, 60%, and 80% with respect to the initial uninformative one. With this process, we essentially built our prior knowledge (with different levels of accuracy) based on past observations. Thus, we executed a testing session for each prior, and each element in $\Lambda \times \mathcal{C}$ for a total of 360 testing sessions.

Fig. 8 shows the total cost measured as the number of tests needed to achieve termination of the learning stage. In particular, Fig. 8(b) shows the cost by grouping the testing sessions by memory configurations. Fig. 8(c) shows the cost grouping by CPU share configurations. Fig. 8(d) shows the cost grouping by replica configurations. Results show that the HDR magnitude affects the total effort. Considering all the configurations, we can observe on average 32k tests when using uninformative priors. When reducing the HDR magnitude by 80% (i.e., highest confidence) we observe on average 17k tests (i.e., total cost reduced by 53%). Considering vertical scaling, we can observe that fewer resources (both memory and CPU) impact negatively on the predictability of the cost. For instance, considering 16 GB memory in Fig. 8(b), the average difference between the 1st and 4th quantiles is 16k. Considering instead 8 GB memory, such a difference increases up to 26k. The same observation holds for Fig. 8(c). For instance, with 50% CPU share, the average difference between the 1st and 4th quantiles is 7k. Considering instead 25% CPU share, the average difference increases up to 26k. The same trend does not hold for horizontal scaling. Here, we observe lower predictability with higher replica values. The average difference between the 1st and 4th quantiles are: 1k with 1 replica, 15k with 2 replicas, and 19k with 4 replicas.

Fig. 9 shows the total cost measured in terms of execution time needed to achieve termination of the learning stage taking into account all the deployment configurations. Consistently with the results shown in Fig. 8, the reduction of the HDR magnitude leads to a decrease in the number of tests and total execution time. Specifically, each testing session lasts on average from ~ 13 to ~ 8 min when reducing the HDR magnitude of the prior knowledge up to 80%.

A comparison between the cost of our approach and the cost of the ground truth baseline (i.e., the DM approach) shows the convenience of our proposal in terms of cost-effectiveness. In particular, as shown in Fig. 8(a) and reported in Avritzer et al. (2020), each testing session that applies the DM assessment requires $\sim 65k$ tests to achieve termination. By running our experiments we observed an average of 51% fewer tests in the worst case (i.e., when using uninformative priors) an average of 73% fewer tests in the best case (i.e., when using accurate priors). As reported in Fig. 9, the execution time required by each testing session is on average reduced by ~ 17 in the worst case and ~ 22 min in the best case.

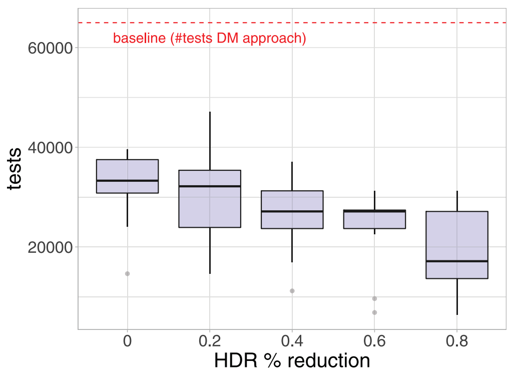
The decreased cost is due to our termination condition driven by the Bayes factor. Namely, it allows each testing session to be stopped as soon as inferred model parameters are statistically stable. As reported in Jiang and Hassan (2015), statistic-driven techniques ensure the accuracy of the collected data and are superior to termination techniques based on pre-defined static configurations (i.e., continuous, timer-driven, and counter-driven) adopted by the DM approach.

RQ3 summary: the prior knowledge has a major impact on the total effort: the smaller the HDR magnitude, the lower the effort. Both vertical and horizontal scaling is likely to affect the predictability of the effort. The comparison between our approach and the selected state-of-the-art baseline shows that we were able to reduce the number of tests up to 51% when using uninformative priors and up to 73% when using accurate priors. The execution time required by each testing session is on average reduced by ~ 17 min in the worst case and ~ 22 min in the best case.

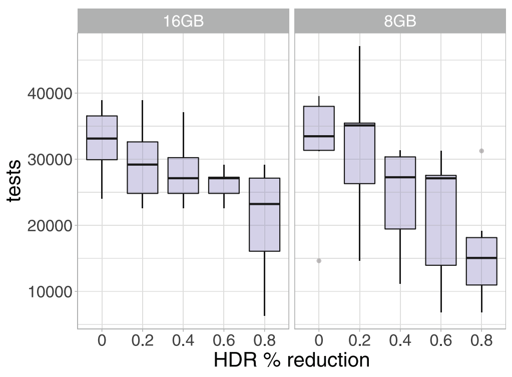
6.3.4. Results for RQ4

What is the cost of the verification process?

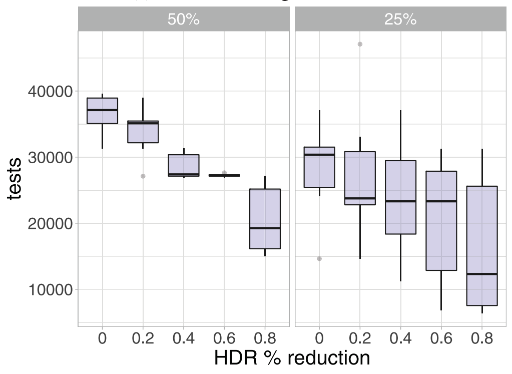
This research question aims at quantifying the total cost required by the verification stage of our methodology. In particular, we focus here on the computational resources (execution time and memory) needed during the verification stage. This phase of our methodology requires the usage of the probabilistic model checker to verify a non-negligible amount of properties: (*i*) system-level CSL requirements; and (*ii*) CSL properties generated from the template $\mathcal{Z}_s(T)$ in Eq. (14). We addressed this question by measuring the resources needed by the PRISM model checker to verify both types of properties. Fig. 10 shows the results of our experiments through bar plots. Figs. 10(a) and 10(b) refer to the verification of system-level requirements reported in Table 3 (execution time and memory consumption per individual configuration, respectively). Each bar in the two plots has been created accounting for the verification of 30 properties (i.e., 5 properties for each $\lambda \in \Lambda$). Considering all the configurations, we measured the resources needed to verify a total of 360 properties. Fig. 10(c)



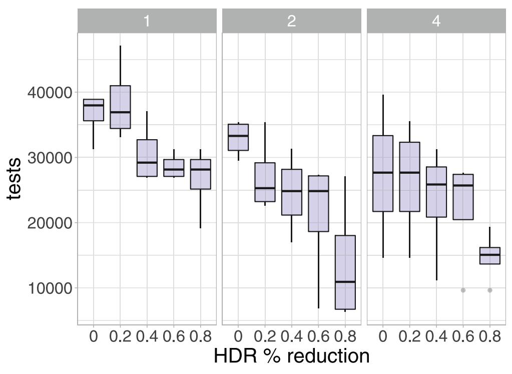
(a) #tests considering all the runs.



(b) #tests per memory configuration.



(c) #tests per CPU share configuration.



(d) #tests per number of replicas.

Fig. 8. Cost in terms of #tests by varying the factors: memory, CPU share, and replicas.

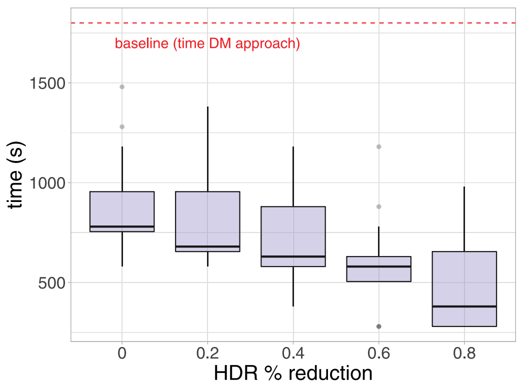


Fig. 9. Cost in terms of execution time of a testing session.

(execution time) and Fig. 10(d) (memory consumption) refer to the verification of the properties needed by the computation of the configuration score φ_λ for each $\lambda \in \Lambda$. In this case, each bar in the two plots has been created accounting for the verification of 5700 properties (i.e., 950 properties for each $\lambda \in \Lambda$), for a total of 68400 properties, considering all the configurations.

Concerning system-level requirements, Fig. 10(a) shows that each CSL property requires at most $\sim$ 1 s for all configurations. Thus, according to our results, the verification of all the requirements for all the configurations took, in our case $\sim$ 1 min. The memory consumption reported in Fig. 10(b) is at most $\sim$ 34 KB per individual requirement and all the configurations. Concerning the configuration score, Fig. 10(c) shows that the time required by the computation of φ_λ is always less than 1 s with a median value less than 0.2 s. Assuming the worst-case scenario where each φ_λ takes 0.8 s, the computation of the total score Φ takes less than 2 min ($\sim$72 s). The memory consumption per deployment configuration is shown in Fig. 10(d). We can observe that the memory required by the computation of φ_λ is always less than 50 KB with a median value less than 3 KB.

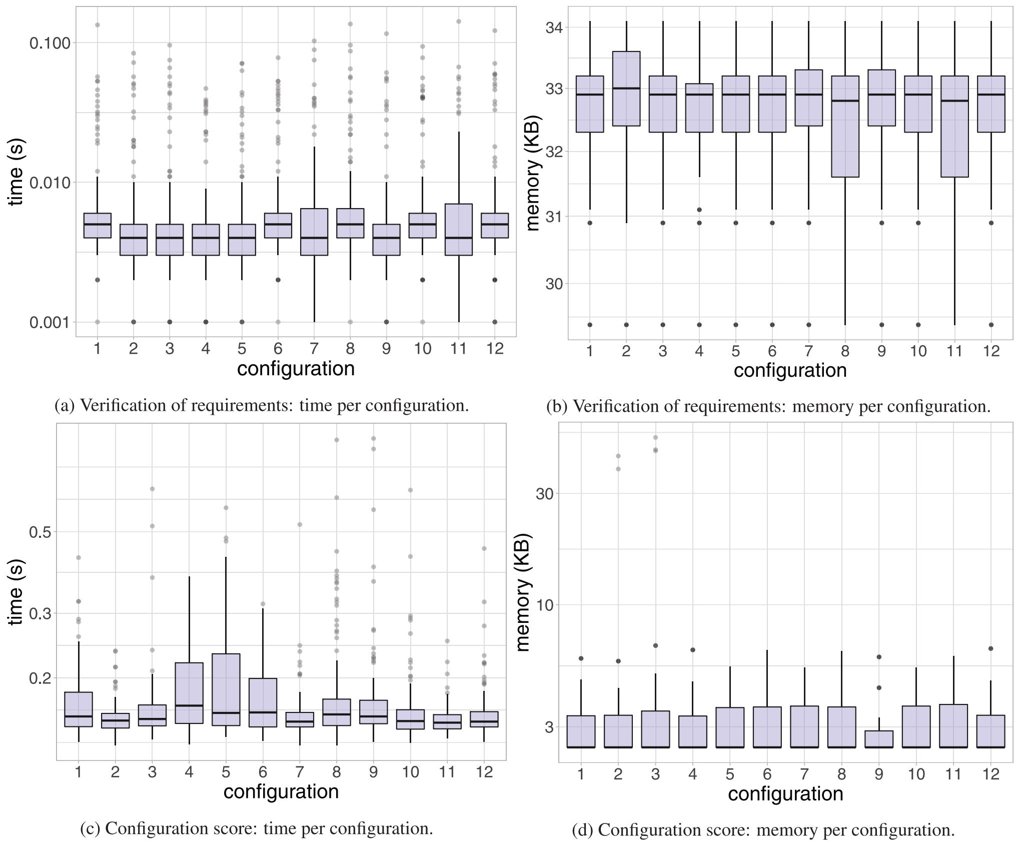
RQ4 summary: the computational resources required by the verification process have been measured by verifying 360 system-level CSL requirements and 68400 CSL properties automatically generated from the meta-property $\mathcal{Z}_s(T)$ to compute φ_λ for all λ. Results show negligible execution time and memory consumption. We observed that the verification of all the requirements for all the configurations took, in our case, less than a minute by consuming at most $\sim$34 KB. The computation of the total score Φ takes $\sim$72 s in the worst case, whereas the worst-case memory consumption is 50 KB.

6.4. Threats to validity

In the empirical evaluation of our methodology, we identified and mitigated major threats to validity in the categories *external*, *internal*, *conclusion*, and *construct* as described in the following.

6.4.1. External validity

Threats to external validity have been addressed by selecting a representative microservices benchmarking system, as anticipated in Section 4. Sock Shop has been recently adopted as SUT by recent scalability assessment approaches tailored to microservices, such as the DM approach (Avritzer et al., 2020). This enabled direct quantitative comparison of the results we obtained during our empirical evaluation, as described in Section 6.3. Our selected SUT also makes use of a common technology stack usually adopted in our target domain of interest. Namely, our in-vitro testing environment supports microservices deployed onto Docker containers running in turn onto virtual machine cores. It



(a) Verification of requirements: time per configuration.
(b) Verification of requirements: memory per configuration.
(c) Configuration score: time per configuration.
(d) Configuration score: memory per configuration.

Fig. 10. Cost of the verification stage in terms of resources (time and memory).

is worth noting that our approach needs to work with a pre-production deployment where we can have full control over the testing environment and the deployment configuration. This setting is common in DevOps practices and tools in modern infrastructure that support continuous deployment (Bertolino et al., 2020). Additional generalization of our findings in other application domains requires experimental activities with a diverse set of case studies.

6.4.2. *Internal validity*

To reduce threats to internal validity, we designed a number of controlled experiments (to assess both the model learning and the verification stages) by detailing the independent variables of interest we controlled during the experiments. In particular, our in-vitro testing environment, as well as the software toolchain implementation of the main stages, allow for direct access to: distribution of the workload intensity, usage profiles, behavior mix, deployment configuration, and prior knowledge. This direct manipulation has been fundamental to assess cause–effect relations between external factors and both benefits and costs of our approach.

6.4.3. *Conclusion validity*

Since testing sessions are guided by a stochastic sampling of user categories, there exists the possibility that results have been produced by chance. We addressed this threat to conclusion validity by following the practical guidelines introduced in Arcuri and Briand (2011). In particular, we sampled a large number of invocations per individual microservice. Each individual testing session, depending on the prior knowledge, sampled between ∼5k and ∼50k requests. Furthermore, we assessed the statistical stability of the inferred CTMC parameters using the Bayes factor, as detailed in Section 5. During the evaluation of the verification stage, we adopted the PRISM model checker to verify a large number of CSL properties generated from the meta-property $\mathcal{Z}_s(T)$. Namely, our conclusion on the cost (time and memory) has been drawn by observing the verification of 68400 properties to compute the total configuration score.

6.4.4. *Construct validity*

We addressed major construct validity threats by assessing the metrics used during our experiments to measure both the effectiveness and the cost of our methodology. The cost of each testing session has been measured by considering the number of tests (i.e., issued requests in our case) and the execution time required to achieve termination. These metrics are often adopted to assess testing methods as described in Arcuri and Briand (2011). The cost of the verification stage has been measured by accounting for execution time and memory consumption. Again, these metrics represent a de facto standard to measure the performance of model checking activities, as described in Katoen et al. (2001). The effectiveness of the model learning stage has been measured by collecting verification errors and then by assessing the ability to minimize them. To measure the confidence of the inference process, we adopted the HDR magnitude which yields the highest possible accuracy in estimating the parameters of a stochastic phenomenon of interest. As described in Robert (2007), this measure is traditionally accepted in Bayesian statistics as a sensible metric.

The applicability of our approach highly depends on the accuracy of the target operational setting under consideration. Therefore, we consider a careful analysis of the production usage as a precondition of the application of our methodology. As discussed in Vögele et al. (2018), automated techniques can be used to deal with this threat. Namely, DTMC behavioral models describing the users can be extracted from recorded session logs of production systems using unsupervised learning, such as clustering algorithms. Therefore, while some specific results on the assessment of Sock Shop depend on the settings described in Section 5.1, the insights and the discussion of benefits and costs of the core stages of our methodology can be generalized.

7. Related work

To the best of our knowledge, our work introduces the first tool-supported methodology for inferring and verifying CTMC models of microservices systems out of automated load testing sessions. We discuss the related work according to the three key aspects upon which our approach builds: load testing, model learning, and verification of microservices systems.

7.1. Load testing

Existing load testing approaches focus on either the detection of functional problems that manifest themselves at certain workload intensities (e.g., deadlocks, race conditions, memory leaks) or violations of non-functional quality-related requirements under loads (e.g., reliability, robustness, performance) (Avritzer et al., 2018; Jiang and Hassan, 2015; Avritzer and Weyuker, 1995). In either case, load testing is typically executed by simulating the users' behavior at different loads (Avritzer et al., 2018, 2020). To this aim, Application Performance Management (APM) tools (Ahmed et al., 2016) are commonly used in practice to mine operational data and extract representative workload models. While most of these approaches focus on system-level testing, algorithms for testing single microservices and their integration are recently emerging to fit modern development practices (Schulz et al., 2019). In our methodology, we execute load testing sessions to learn a CTMC model and then carry out formal verification of system-level performance requirements. To this end, we define load testing objectives formally based on the user needs by encoding requirements as system-level CSL properties crafted by engineers during the early design stages. As described in Jiang and Hassan (2015) and Ho et al. (2008), such objectives are not always defined in a rigorous way and they may also emerge late, after an initial observation period, by establishing the baseline performance of a software version, which is then used to compare the load testing results of later software versions (Eismann et al., 2020).

Quantifying performance requirements is the basic building block of the Domain Metric approach (Avritzer et al., 2018, 2020) that we have used as a baseline for comparison in our evaluation. The pass/fail criterion of testing sessions presented in this work is extracted empirically (Pozin and Galakhov, 2011) by observing the system under specific low load conditions and then extracting a performance threshold that defines outlier microservices. Differently from our approach, this work assumes that under low load, the system fulfills the requirements. Under this assumption, the average performance under increasing workload intensity is then evaluated against such a threshold. This approach is particularly useful in exploratory studies where performance requirements are not available (Menascé and AF, 2002). The performance score adopted by our methodology extends the baseline Domain Metric introduced in Avritzer et al. (2018) by computing, for each microservice, the likelihood of observing a response time in between a lower-bound and an upper-bound defined on human perceptual abilities (Nielsen, 1994).

The use of Markov models to drive load testing of service-based and web applications is not new (Menascé et al., 1999; Mark and Csaba, 2007). The authors introduce the notion of Customer Behavior Model Graphs, which is a DTMC enriched with domain-specific aspects. Similar to our approach, user sessions are used to identify the states in the Markov model (i.e., user interaction with the system such as service requests), and transitions between these states, annotated with probabilities. The WESSBAS approach adopts workload characterization and extraction by combining DTMC and Finite State Machine models (Vögele et al., 2018). Our method captures the behavior of users and then drives the load testing sessions through the notion of behavior mix which identifies and mixes different user categories defined as sequences of valid requests to available microservices. As described in Menascé et al. (1999) and Vögele et al. (2018), clustering algorithms can be conveniently leveraged to automate the identification of different customer groups and then create the user categories (Menascé et al., 1999).

Another key aspect characterizing load testing sessions is the termination condition. The termination is usually decided by a controller component belonging to the load driver node which coordinates one or more distributed nodes where the SUT is deployed (Dumitrescu et al., 2004). Termination techniques based on pre-defined configurations, such as timer-driven and counter-driven are common in many existing load testing approaches (Stankovic, 2006). Our approach is based on the notion of statistic-driven termination that was introduced to overcome the shortcomings of these approaches and ensure the validity or accuracy of the data (Snellman et al., 2011). This means that a load testing session terminates when the performance metrics of interest are statistically stable according to a high confidence interval or a small standard deviation as described in Snellman et al. (2011) and Mansharamani et al. (2010). Our approach differs from these available techniques since it is based on the convergence of the Bayes factor that detects when the inferred parameters of the CTMC model are statistically stable and therefore, further refinement is not needed.

7.2. Model learning

Model learning techniques have been receiving increased attention from the software engineering community (Aichernig et al., 2018). These techniques are usually categorized as either *passive* or *active*. Passive learning mines historical datasets such as available system logs. Active learning queries the SUT by means of suitable testing activities and is usually considered more efficient, (Smeenk et al., 2015) and it is referred to as test-based learning. Our approach falls in this category and is inspired by recent literature that leverages probabilistic methods to model and reason in a quantitative way about QoS attributes of the target system (Zhang et al., 2019; Perez-Palacin and Mirandola, 2014; Camilli et al., 2018; Camilli et al., 2021). In this context, the usage of Markov models is a common choice justified by the practical need of approximating stochastic phenomena of interest (e.g., performance, reliability). The METRIC approach introduced in Camilli et al. (2017, 2020) applies a test-based learning technique to derive a full specification of the SUT as Markov Decision Processes to increase the level of assurance by verifying system-level requirements after accounting for runtime evidence. From this perspective, our methodology is conceptually similar, since it leverages testing to derive a model which is then verified against formal requirements. Nevertheless, the major objective of METRIC is different since it focuses on reliability aspects that may be affected by uncertain environmental factors like the failure of third-party services.

Continuous inference of categorical distributions has been proposed in Filieri et al. (2016, 2015), to learn transition probabilities of a DTMC model. Like our methodology, these latter approaches adopt Bayesian inference, but they use aging mechanisms (e.g., Kalman filters Astrom and Wittenmark, 1990) to discard old information and rapidly detect occurring changes. This method is particularly suitable for continuous inference of QoS attributes in production (rather than testing) to cope with evolving usage behaviors and environmental conditions. Other Markov models, such as Hierarchical Hidden Markov Models (Fine et al., 1998) have been recently adopted. For instance, the DLA (Detecting and Localizing Anomalies) framework introduced in Samir and Pahl (2019) uses different workload intensity bins to test and learn the behavior of the target system as in our approach. By contrast, DLA detects the variation in response time and correlates it to the respective component (microservice, container, service), which yields anomalous behavior. In this case, the ultimate goal is to detect and locate the anomalies, such as CPU Hog, memory leak, and network congestion. Efficiency analysis of microservices provisioning based on a CTMC modeling approach has been introduced in Khazaei et al. (2016). Even though our methodology adopts the same modeling formalism, the approach and the main goals are different. The work presented in Khazaei et al. (2016) targets Infrastructure-as-a-Service cloud providers in order to model container and virtual machine provisioning and request arrival rate over time. Differently, our methodology focuses on the inference of the probability density functions describing the response time associated with each individual microservice. The density functions are then encoded as parameters of the CTMC models adopting the standard assumption that service execution times are exponentially distributed. A similar approach has been recently adopted by the framework OMNI (Paterson and Calinescu, 2020). Like our methodology, OMNI supports learning and verification of system-level QoS requirements, but it uses phase-type distributions (PHDs) to refine the relevant elements of a CTMC model. Although PHDs can arbitrarily close approximate any continuous distribution with a strictly positive density, (O'cinneide, 1999), fitting a PHD to heavy-tailed distributions requires extremely large samples and it is generally more expensive than applying the Bayesian updating rule adopted by our approach.

7.3. Verification of microservices systems

Behavioral types, choreographies, refinement types, and other (semi-) formal behavioral models have been proposed to specify and verify large distributed service-based systems. The research community is still active on these topics and much has to be discovered and developed to support microservices engineering (Dragoni et al., 2017). Formal methods based on well-known techniques represent a promising approach for tackling the challenges of delivering microservices systems in compliance with rigorous requirements. Nevertheless, how exactly these disciplines can be extended to naturally capture the practical needs in the domain of microservices requires further investigation as advocated by the work presented in Dragoni et al. (2017) and Camilli (2020). The approach introduced in Wadler (2012) is grounded on logical reconstruction of behavioral types in classical linear logic. Another logical characterization of choreography-based behavioral types has been proposed in Carbone et al. (2017). This latter approach aims at specifying and verifying existing interactions amongst multiple microservices using well-known techniques for logical reasoning (Caires and Pérez, 2016). A model-driven engineering approach to support microservices engineering, through formal specification and verification activities using Petri Nets, has been recently introduced in Camilli (2020) and Camilli et al. (2018b,a). These approaches are tailored to design-time specification and analysis of the expected behavior of target microservices. Thus, they lack feedback from runtime evidence to learn or refine the initial specification of the system. As such, our methodology enhances QoS analysis through the learning process that can prevent invalid engineering decisions based on design-time CTMC analysis. This approach is inspired by existing software performance engineering methods used to decide a feasible service-level agreement after collecting runtime evidence (Woodside et al., 2007). A similar perspective is adopted by the OMNI approach (Paterson and Calinescu, 2020). OMNI requires the collection of component observations either by testing the intended system components (or services), or by monitoring other systems that use these components. The refined CTMC models generated by OMNI are then analyzed with probabilistic model checkers, such as PRISM (Kwiatkowska et al., 2011), also adopted by our software toolchain. Both OMNI and our methodology support the verification of a broad spectrum of system-level performance requirements specified in CSL (Baier et al., 2003).

8. Discussion

Microservices systems are calling for novel quality assurance approaches that, on the one hand, can seamlessly be integrated into modern development practices and, on the other hand, provide strong, ideally provable assurances. To this aim, we have introduced a novel framework for test-based learning and verification of microservices systems. In this section, we discuss the advantages and limitations of our framework to pave the way for further research.

8.1. Strengths

Rigorous foundation. Interaction between development and operations emerged in the software industry as an important principle while supporting the engineering life-cycle of microservices systems. These phases are often interleaved and should lead to incremental quality improvements to better fit customer needs. Our experience shows how information extracted from operations can drive automated test-based learning and verification processes supported by off-the-shelf tools. This means that formal aspects are hidden from engineers operating the toolchain. The DTMC model driving the testing sessions is automatically derived from the definition of the user categories. The testing process feeds the inference module that builds the CTMC models by applying fully automated machinery. Then, the model checker is used as a black box to automatically verify the CTMC models and obtain the configuration score. Our work contributes to the advance of common performance testing practices with rigorous methods to ultimately achieve high quality with strong guarantees.

Strong assurances with limited cost. State of the art model checkers, such as PRISM (Kwiatkowska et al., 2011), and Storm (Dehnert et al., 2017) have been continuously improved and extended with efficient analysis techniques, like symbolic approaches (Katoen et al., 2001) that yield fast and scalable verification of systems whose state space has an order of magnitude up to 10^6 or 10^7. By contrast, the number of microservices of real-world applications is relatively smaller. As an example, Netflix recently declared that, even though the total number of microservices in their systems is around 500, their workflows are made of 6 services on average, with the largest one composed of 48 services (Netflix, 2016). In our methodology, the number of microservices determines the structural complexity (i.e., number of states and transitions) of the CTMC models that are then verified against system-level requirements. Since the scale of operation of state-of-the-art model checkers is much higher, this represents a great opportunity to achieve strong assurances with a limited cost, as shown by our evaluation.

Requirements traceability. Our approach allows engineers to follow the life of system-level requirements in both forward and reverse directions. In particular, they define meaningful CSL properties according to the business goals of the target application. Then, automated verification traces performance issues back to unfulfilled requirements. This allows engineers to spot workflows (and related microservices) responsible for the (un)achievement of specific goals that are relevant for the business logic of the target system.

Guidance at decision gates. The configuration score provides engineers with additional insights. Given a number of configurations satisfying the requirements, engineers can choose and deploy the one associated with the highest score that yields the best fitness according to the target operational setting. Furthermore, our experience shows that by comparing the scores, we can understand how parameters of the configuration are likely to affect the fitness. According to our results, we understood that vertical resources of our in-vitro setting are (in some cases) not enough to scale horizontally with a number of service replicas greater than specific thresholds.

8.2. Limitations

Nontrivial definition of accurate priors. Although our experience shows that uninformative priors lead to outperforming existing methods, leveraging the prior knowledge can help in reducing, even more, the overall cost of the testing process by limiting the number of tests required to achieve termination. Transferring the prior knowledge into accurate prior density functions may not be perceived as straightforward since this activity requires skills in probabilistic methods. In this case, tools for interactive (machine-assisted) elicitation of prior density functions may be adopted to mitigate this issue (Morris et al., 2014).

Combinatorial explosion of configurations. The cost of testing all possible execution contexts for a target system is subject to a combinatorial explosion. At the current stage, our methodology does not explicitly deal with this issue. For this reason, the design of the domains Λ (workload intensity) and $\mathcal{C}$ (deployment configurations) requires particular attention to limit the total amount of load testing sessions. In our experiments, we aggregated the workload intensity by using data binning. We also limited the number of deployment configurations to be tested exploiting our domain knowledge. We focused on discrete sets of CPU percentage values, amount of memory, and the number of instances of a single microservice `cart`, which is the one associated with the highest steady-state probability according to the DTMC model. When no domain knowledge is available, engineers may follow, for instance, coverage-oriented (Siegmund et al., 2012) or distance-based sampling (Kaltenecker et al., 2019) strategies to select a small representative sample of configurations to be tested.

Missing information or unexpected changes. The outcome of the verification stage heavily relies on the rigorous definition of the user categories and the behavior mix that reflect our assumptions on the execution context. A cold start might imply that such pieces of information are missing. In this case, the DTMC model may represent an initial guess (e.g., based on past experience with similar systems) and therefore, the overall usefulness of the approach inevitably decreases. Furthermore, sudden and unexpected changes in the usage profile, as well as workload intensity, can lead to situations that have not been tested. For this reason, continuous monitoring of the system in production is necessary to detect such changes and then run again the core stages in the forthcoming development cycles. This practice is becoming more and more common since the cost of an exhaustive exploration of all possible scenarios and execution contexts is too high and it does not match the agile attitude typically applied in modern engineering life-cycles. Thus, continuous monitoring shall complement the core stages of our methodology to learn from the history of recent executions and improve our knowledge on the usage profile that is prone to change dynamically.

9. Conclusion

In this paper, we introduced a novel approach to test-based model learning and verification of microservices systems under alternative deployment configurations. Our methodology combines specification-driven load testing sessions and Bayesian inference to learn a performance model of the target microservices system in terms of a CTMC. The CTMC formal description is then used to verify CSL requirements and rank the available deployment configurations based on the notion of configuration score that aims at evaluating how well the SUT fits the expected operational setting under increasing workload intensity. The whole approach is supported by a software toolchain implementation that has been released publicly to encourage the adoption and repetition of experiments.

Our evaluation conducted on a representative microservices system benchmark for the research community, shows major benefits, costs, and threats to validity of the core stages of our methodology. We have shown that rigorous engineering methods can be used to improve the cost-effectiveness of performance assessment methods for microservices systems. In particular, the model learning automatically terminates the testing process as soon as inferred model parameters are statistically stable to avoid verification errors and, at the same time, reduce the total cost. A quantitative comparison between our approach and the Domain Metric baseline (i.e., an existing state-of-the-art scalability assessment approach) shows that we were able to reduce the number of tests up to 51% when using uninformative priors and up to 73% when using accurate priors. The execution time required by each testing session is on average reduced by $\sim$17 and $\sim$22 min, respectively.

Our experience in operating our methodology shows that it can be successfully applied to spot performance issues and trace them back to requirements in order to support automated decision gates. The configuration score complements the verification of requirements by identifying relations between configurations parameters and the overall performance of the system. For instance, we found that the physical constraints of our in-vitro environment limited the ability to apply horizontal scaling. Specifically, we observed a performance degradation when more than 2 microservice replicas were adopted with less than 50% CPU share. Both the verification of requirements and the computation of the configuration score are supported by off-the-shelf tools that require negligible execution time and memory consumption. In our experiments, we observed that the verification of 360 system-level CSL requirements took $\sim$ 1 min by consuming at most 34 KB. The computation of the total score involved the verification of 68400 (automatically generated) CSL properties that required $\sim$72 s and at most $\sim$50 KB.

We plan to extend the methodology by adding a pre-processing stage by using parametric model checking (Hahn et al., 2010). The pre-processing aims at identifying preconditions on the variability of the parameters in the CTMC model that meet the satisfaction of the system-level requirements. As described in Camilli and Russo (2020), this latter approach has the potential of further reducing the cost of the testing process by terminating it as soon as violations are verified at runtime.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was supported by the "Open Access Publishing Fund" of the Free University of Bozen-Bolzano.

References

Ahmed, T.M., Bezemer, C.-P., Chen, T.-H., Hassan, A.E., Shang, W., 2016. Studying the effectiveness of application performance management (APM) tools for detecting performance regressions for web applications: An experience report. In: Proceedings of the 13th International Conference on Mining Software Repositories. MSR '16, Association for Computing Machinery, New York, NY, USA, pp. 1–12. http://dx.doi.org/10.1145/2901739.2901774.

Aichernig, B.K., Mostowski, W., Mousavi, M.R., Tappler, M., Taromirad, M., 2018. Model learning and model-based testing. In: Bennaceur, A., Hähnle, R., Meinke, K. (Eds.), Machine Learning for Dynamic Software Analysis: Potentials and Limits: International Dagstuhl Seminar 16172, Dagstuhl Castle, Germany, April 24-27, 2016, Revised Papers. Springer International Publishing, Cham, pp. 74–100. http://dx.doi.org/10.1007/978-3-319-96562-8_3.

Alshuqayran, N., Ali, N., Evans, R., 2016. A systematic mapping study in microservice architecture. In: Proc. IEEE 9th International Conference on Service-Oriented Computing and Applications, SOCA 2016. pp. 44–51.

Anderson, W.J., 2012. Continuous-Time Markov Chains: An Applications-Oriented Approach. Springer, New York, NY, http://dx.doi.org/10.1007/978-1-4612-3038-0.

Arcuri, A., Briand, L., 2011. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In: Proceedings of the 33rd International Conference on Software Engineering. ICSE '11, Association for Computing Machinery, New York, NY, USA, pp. 1–10. http://dx.doi.org/10.1145/1985793.1985795.

Assunção, W.K.G., Krüger, J., Mendonça, W.D.F., 2020. Variability management meets microservices: Six challenges of re-engineering microservice-based webshops. SPLC '20, In: Proceedings of the 24th ACM Conference on Systems and Software Product Line, vol. A, Association for Computing Machinery, New York, NY, USA, http://dx.doi.org/10.1145/3382025.3414942.

Astrom, K.J., Wittenmark, B., 1990. Computer-Controlled Systems: Theory and Design, second ed. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Autili, M., Grunske, L., Lumpe, M., Pelliccione, P., Tang, A., 2015. Aligning qualitative, real-time, and probabilistic property specification patterns using a structured english grammar. IEEE Trans. Softw. Eng. 41 (7), 620–638. http://dx.doi.org/10.1109/TSE.2015.2398877.

Avritzer, A., Britto, R., Trubiani, C., Russo, B., Janes, A., Camilli, M., van Hoorn, A., Heinrich, R., Rapp, M., Henß, J., 2021a. A multivariate characterization and detection of software performance antipatterns. In: Proceedings of the ACM/SPEC International Conference on Performance Engineering. ICPE '21, Association for Computing Machinery, New York, NY, USA, pp. 61–72. http://dx.doi.org/10.1145/3427921.3450246.

Avritzer, A., Camilli, M., Janes, A., Russo, B., Jahič, J., Hoorn, A.v., Britto, R., Trubiani, C., 2021b. PPTAM$^{\lambda}$: What, where, and how of cross-domain scalability assessment. In: 2021 IEEE 18th International Conference on Software Architecture Companion, ICSA-C. pp. 62–69. http://dx.doi.org/10.1109/ICSA-C52384.2021.00016.

Avritzer, A., Ferme, V., Janes, A., Russo, B., van Hoorn, A., Schulz, H., Menasché, D., Rufino, V., 2020. Scalability assessment of microservice architecture deployment configurations: A domain-based approach leveraging operational profiles and load tests. J. Syst. Softw. 165, 110564. http://dx.doi.org/10.1016/j.jss.2020.110564.

Avritzer, A., Ferme, V., Janes, A., Russo, B., Schulz, H., van Hoorn, A., 2018. A quantitative approach for the assessment of microservice architecture deployment alternatives by automated performance testing. In: Proceedings of the 12th European Conference on Software Architecture, ECSA. pp. 159–174, http://dx.doi.org/10.1007/978-3-030-00761-4_11.

Avritzer, A., Weyuker, E.R., 1995. The automatic generation of load test suites and the assessment of the resulting software. IEEE Trans. Softw. Eng. 21 (9), 705–716. http://dx.doi.org/10.1109/32.464549.

Baier, C., Haverkort, B., Hermanns, H., Katoen, J., 2003. Model-checking algorithms for continuous-time Markov chains. IEEE Trans. Softw. Eng. 29 (6), 524–541. http://dx.doi.org/10.1109/TSE.2003.1205180.

Berger, J.O., 1985. Statistical Decision Theory and Bayesian Analysis. In: Springer Series in Statistics, Springer.

Bertolino, A., Angelis, G.D., Guerriero, A., Miranda, B., Pietrantuono, R., Russo, S., DevOpRET: Continuous reliability testing in DevOps. J. Softw. Evol. Process n/a (n/a), e2298. http://dx.doi.org/10.1002/smr.2298, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.2298, e2298 smr.2298,

Bertolino, A., Angelis, G., Guerriero, A., Miranda, B., Pietrantuono, R., Russo, S., 2020. DevOpRET: COntinuous reliability testing in DevOps. J. Softw. Evol. Process http://dx.doi.org/10.1002/smr.2298.

Caires, L., Pérez, J.A., 2016. Multiparty session types within a canonical binary theory, and beyond. In: 36th IFIP WG 6.1 International Conference on Formal Techniques for Distributed Objects, Components, and Systems - Volume 9688. Springer-Verlag, Berlin, Heidelberg, pp. 74–95. http://dx.doi.org/10.1007/978-3-319-39570-8_6.

Camilli, M., 2020. Continuous formal verification of microservice-based process flows. In: Muccini, H., Avgeriou, P., Buhnova, B., Camara, J., Caporuscio, M., Franzago, M., Koziolek, A., Scandurra, P., Trubiani, C., Weyns, D., Zdun, U. (Eds.), Software Architecture. Springer International Publishing, Cham, pp. 420–435. http://dx.doi.org/10.1007/978-3-030-59155-7_31.

Camilli, M., Bellettini, C., Capra, L., 2018a. Design-time to run-time verification of microservices based applications. In: Cerone, A., Roveri, M. (Eds.), Software Engineering and Formal Methods. Springer International Publishing, Cham, pp. 168–173. http://dx.doi.org/10.1007/978-3-319-74781-1_12.

Camilli, M., Bellettini, C., Capra, L., Monga, M., 2018b. A formal framework for specifying and verifying microservices based process flows. In: Cerone, A., Roveri, M. (Eds.), Software Engineering and Formal Methods. Springer International Publishing, Cham, pp. 187–202. http://dx.doi.org/10.1007/978-3-319-74781-1_14.

Camilli, M., Bellettini, C., Gargantini, A., Scandurra, P., 2018. Online model-based testing under uncertainty. In: 2018 IEEE 29th International Symposium on Software Reliability Engineering, ISSRE. pp. 36–46, http://dx.doi.org/10.1109/ISSRE.2018.00015.

Camilli, M., Gargantini, A., Scandurra, P., 2020. Model-based hypothesis testing of uncertain software systems. Softw. Test. Verif. Reliab. 30 (2), e1730. http://dx.doi.org/10.1002/stvr.1730, e1730 stvr.1730. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1730.

Camilli, M., Gargantini, A., Scandurra, P., Bellettini, C., 2017. Towards inverse uncertainty quantification in software development (short paper). In: Cimatti, A., Sirjani, M. (Eds.), Software Engineering and Formal Methods - 15th International Conference, SEFM 2017, Trento, Italy, September 4-8, 2017, Proceedings. In: Lecture Notes in Computer Science, vol. 10469, Springer, pp. 375–381. http://dx.doi.org/10.1007/978-3-319-66197-1_24.

Camilli, M., Gargantini, A., Scandurra, P., Trubiani, C., 2021. Uncertainty-aware exploration in model-based testing. In: 2021 14th IEEE Conference on Software Testing, Verification and Validation, ICST. pp. 71–81. http://dx.doi.org/10.1109/ICST49551.2021.00019.

Camilli, M., Russo, B., 2020. Model-based testing under parametric variability of uncertain beliefs. In: de Boer, F., Cerone, A. (Eds.), Software Engineering and Formal Methods. Springer International Publishing, Cham, pp. 175–192. http://dx.doi.org/10.1007/978-3-030-58768-0_10.

Carbone, M., Montesi, F., Schürmann, C., Yoshida, N., 2017. Multiparty session types as coherence proofs. Acta Inform. 54, http://dx.doi.org/10.1007/s00236-016-0285-y.

Clarke, E.M., Grumberg, O., Peled, D.A., 2000. Model Checking. MIT Press, Cambridge, MA, USA.

Courcoubetis, C., Yannakakis, M., 1995. The complexity of probabilistic verification. J. ACM 42 (4), 857–907. http://dx.doi.org/10.1145/210332.210339.

Dehnert, C., Junges, S., Katoen, J.-P., Volk, M., 2017. A storm is coming: A modern probabilistic model checker. In: Majumdar, R., Kunčak, V. (Eds.), Computer Aided Verification. Springer International Publishing, Cham, pp. 592–600. http://dx.doi.org/10.1007/978-3-319-63390-9_31.

Diaconis, P., Ylvisaker, D., et al., 1979. Conjugate priors for exponential families. Ann. Statist. 7 (2), 269–281.

Dragoni, N., Giallorenzo, S., Lafuente, A.L., Mazzara, M., Montesi, F., Mustafin, R., Safina, L., 2017. Microservices: Yesterday, today, and tomorrow. In: Mazzara, M., Meyer, B. (Eds.), Present and Ulterior Software Engineering. Springer International Publishing, Cham, pp. 195–216. http://dx.doi.org/10.1007/978-3-319-67425-4_12.

Dumitrescu, C., Raicu, I., Ripeanu, M., Foster, I., 2004. Diperf: An automated distributed performance testing framework. In: Fifth IEEE/ACM International Workshop on Grid Computing. pp. 289–296. http://dx.doi.org/10.1109/GRID.2004.21.

Eismann, S., Bezemer, C.-P., Shang, W., Okanović, D., van Hoorn, A., 2020. Microservices: A performance tester's dream or nightmare? In: Proceedings of the ACM/SPEC International Conference on Performance Engineering. ICPE '20, Association for Computing Machinery, New York, NY, USA, pp. 138–149. http://dx.doi.org/10.1145/3358960.3379124.

Ferme, V., Pautasso, C., 2018. A declarative approach for performance tests execution in continuous software development environments. In: Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering. ICPE '18, Association for Computing Machinery, New York, NY, USA, pp. 261–272. http://dx.doi.org/10.1145/3184407.3184417.

Filieri, A., Ghezzi, C., Tamburrelli, G., 2012. A formal approach to adaptive software: Continuous assurance of non-functional requirements. Form. Asp. Comput. 24 (2), 163–186. http://dx.doi.org/10.1007/s00165-011-0207-2, URL http://dx.doi.org/10.1007/s00165-011-0207-2.

Filieri, A., Grunske, L., Leva, A., 2015. Lightweight adaptive filtering for efficient learning and updating of probabilistic models. ICSE '15, In: Proceedings of the 37th International Conference on Software Engineering, vol. 1, IEEE Press, Piscataway, NJ, USA, pp. 200–211, URL http://dl.acm.org/citation.cfm?id=2818754.2818781.

Filieri, A., Tamburrelli, G., Ghezzi, C., 2016. Supporting self-adaptation via quantitative verification and sensitivity analysis at run time. IEEE Trans. Softw. Eng. 42 (1), 75–99. http://dx.doi.org/10.1109/TSE.2015.2421318.

Fine, S., Singer, Y., Tishby, N., 1998. The hierarchical hidden Markov model: Analysis and applications. Mach. Learn. 32 (1), 41–62. http://dx.doi.org/10.1023/A:1007469218079.

Forejt, V., Kwiatkowska, M., Norman, G., Parker, D., 2011. Automated verification techniques for probabilistic systems. In: Formal Methods for Eternal Networked Software Systems: 11th International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM 2011, Bertinoro, Italy, June 13-18, 2011. Advanced Lectures. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 53–113. http://dx.doi.org/10.1007/978-3-642-21455-4_3.

Ghezzi, C., 2018. Formal methods and agile development: Towards a happy marriage. In: Gruhn, V., Striemer, R. (Eds.), The Essence of Software Engineering. Springer International Publishing, Cham, pp. 25–36. http://dx.doi.org/10.1007/978-3-319-73897-0_2.

Grambow, M., Meusel, L., Wittern, E., Bermbach, D., 2020. Benchmarking microservice performance: A pattern-based approach. In: Proceedings of the 35th Annual ACM Symposium on Applied Computing. SAC '20, Association for Computing Machinery, New York, NY, USA, pp. 232–241. http://dx.doi.org/10.1145/3341105.3373875.

Grunske, L., 2008. Specification patterns for probabilistic quality properties. In: 2008 ACM/IEEE 30th International Conference on Software Engineering. pp. 31–40. http://dx.doi.org/10.1145/1368088.1368094.

Guerriero, A., Mirandola, R., Pietrantuono, R., Russo, S., 2019. A hybrid framework for web services reliability and performance assessment. In: 2019 IEEE International Symposium on Software Reliability Engineering Workshops, ISSREW. pp. 185–192. http://dx.doi.org/10.1109/ISSREW.2019.00070.

Hahn, E.M., Hermanns, H., Wachter, B., Zhang, L., 2010. Param: A model checker for parametric Markov models. In: Touili, T., Cook, B., Jackson, P. (Eds.), Computer Aided Verification. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 660–664. http://dx.doi.org/10.1007/978-3-642-14295-6_56.

Heinrich, R., van Hoorn, A., Knoche, H., Li, F., Lwakatare, L.E., Pahl, C., Schulte, S., Wettinger, J., 2017. Performance engineering for microservices: Research challenges and directions. In: Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion. ICPE '17 Companion, Association for Computing Machinery, New York, NY, USA, pp. 223–226. http://dx.doi.org/10.1145/3053600.3053653.

Ho, C.-W., Williams, L., Robinson, B., 2008. Examining the relationships between performance requirements and "not a problem" defect reports. In: 2008 16th IEEE International Requirements Engineering Conference. pp. 135–144. http://dx.doi.org/10.1109/RE.2008.51.

Insua, D., Ruggeri, F., Wiper, M., 2012. Bayesian Analysis of Stochastic Process Models. In: Wiley Series in Probability and Statistics, Wiley, http://dx.doi.org/10.1002/9780470975916.

Jeffreys, H., 1998. The Theory of Probability. OUP Oxford.

Jiang, Z.M., Hassan, A.E., 2015. A survey on load testing of large-scale software systems. IEEE Trans. Softw. Eng. 41 (11), 1091–1118. http://dx.doi.org/10.1109/TSE.2015.2445340.

Kaltenecker, C., Grebhahn, A., Siegmund, N., Guo, J., Apel, S., 2019. Distance-based sampling of software configuration spaces. In: Proceedings of the 41st International Conference on Software Engineering. ICSE '19, IEEE Press, pp. 1084–1094. http://dx.doi.org/10.1109/ICSE.2019.00112.

Katoen, J.-P., Kwiatkowska, M., Norman, G., Parker, D., 2001. Faster and symbolic CTMC model checking. In: de Alfaro, L., Gilmore, S. (Eds.), Process Algebra and Probabilistic Methods. Performance Modelling and Verification. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 23–38. http://dx.doi.org/10.1007/3-540-44804-7_2.

Khazaei, H., Barna, C., Beigi-Mohammadi, N., Litoiu, M., 2016. Efficiency analysis of provisioning microservices. In: 2016 IEEE International Conference on Cloud Computing Technology and Science, CloudCom. pp. 261–268. http://dx.doi.org/10.1109/CloudCom.2016.0051.

v. Kistowski, J., Eismann, S., Grohmann, J., Schmitt, N., Bauer, A., Kounev, S., 2019. Teastore - a micro-service reference application for performance engineers. In: Companion of the 2019 ACM/SPEC International Conference on Performance Engineering. ICPE '19, Association for Computing Machinery, New York, NY, USA, pp. 47–48. http://dx.doi.org/10.1145/3302541.3311966.

Kistowski, J.v., Herbst, N.R., Kounev, S., 2014. Modeling variations in load intensity over time. In: Proceedings of the Third International Workshop on Large Scale Testing. LT '14, Association for Computing Machinery, New York, NY, USA, pp. 1–4. http://dx.doi.org/10.1145/2577036.2577037.

Kwiatkowska, M., Norman, G., Parker, D., 2011. PRISM 4.0: Verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (Eds.), Proc. 23rd International Conference on Computer Aided Verification, CAV'11. In: LNCS, vol. 6806, Springer, pp. 585–591. http://dx.doi.org/10.1007/978-3-642-22110-1_47.

Mansharamani, R., Khanapurkar, A., Mathew, B., Subramanyan, R., 2010. Performance testing: Far from steady state. In: 2010 IEEE 34th Annual Computer Software and Applications Conference Workshops. pp. 341–346. http://dx.doi.org/10.1109/COMPSACW.2010.66.

Mark, K., Csaba, L., 2007. Analyzing customer behavior model graph (CBMG) using Markov chains. In: 2007 11th International Conference on Intelligent Engineering Systems. pp. 71–76. http://dx.doi.org/10.1109/INES.2007.4283675.

Menascé, D.A., AF, V., 2002. Capacity Planning for Web Services: Metrics, Models, and Methods. Prentice Hall.

Menascé, D.A., Almeida, V.A.F., Fonseca, R., Mendes, M.A., 1999. A methodology for workload characterization of E-commerce sites. In: Proceedings of the 1st ACM Conference on Electronic Commerce. EC '99, Association for Computing Machinery, New York, NY, USA, pp. 119–128. http://dx.doi.org/10.1145/336992.337024.

Morris, D.E., Oakley, J.E., Crowe, J.A., 2014. A web-based tool for eliciting probability distributions from experts. Environ. Model. Softw. 52, 1–4. http://dx.doi.org/10.1016/j.envsoft.2013.10.010, URL https://www.sciencedirect.com/science/article/pii/S1364815213002533.

Netflix, 2016. Netflix conductor: A microservices orchestrator. https://netflixtechblog.com/netflix-conductor-a-microservices-orchestrator-2e8d4771bf40. Online: (Accessed Feb 2021).

Nielsen, J., 1994. Usability Engineering. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Norris, J.R., 1998. Markov Chains. In: Cambridge series in statistical and probabilistic mathematics, Cambridge University Press.

O'cinneide, C.A., 1999. Phase-type distributions: Open problems and a few properties. Stoch. Models 15 (4), 731–757. http://dx.doi.org/10.1080/15326349908807560.

Paterson, C., Calinescu, R., 2020. Observation-enhanced QoS analysis of component-based systems. IEEE Trans. Softw. Eng. 46 (05), 526–548. http://dx.doi.org/10.1109/TSE.2018.2864159.

Perez-Palacin, D., Mirandola, R., 2014. Uncertainties in the modeling of self-adaptive systems: A taxonomy and an example of availability evaluation. In: International Conference on Performance Engineering. pp. 3–14.

Pozin, B.A., Galakhov, I.V., 2011. Models in performance testing. Program. Comput. Softw. 37 (1), 15–25. http://dx.doi.org/10.1134/S036176881101004X.

Robert, C.P., 2007. the Bayesian Choice: From Decision-Theoretic Foundations to Computational Implementation, second ed. Springer.

Samir, A., Pahl, C., 2019. DLA: Detecting and localizing anomalies in containerized microservice architectures using Markov models. In: 2019 7th International Conference on Future Internet of Things and Cloud, FiCloud. pp. 205–213. http://dx.doi.org/10.1109/FiCloud.2019.00036.

Schulz, H., Angerstein, T., Okanović, D., van Hoorn, A., 2019. Microservice-tailored generation of session-based workload models for representative load testing. In: 2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, MASCOTS. pp. 323–335. http://dx.doi.org/10.1109/MASCOTS.2019.00043.

Siegmund, N., Kolesnikov, S.S., Kästner, C., Apel, S., Batory, D., Rosenmüller, M., Saake, G., 2012. Predicting performance via automated feature-interaction detection. In: Proceedings of the 34th International Conference on Software Engineering. ICSE '12, IEEE Press, pp. 167–177. http://dx.doi.org/10.1109/ICSE.2012.6227196.

Smeenk, W., Moerman, J., Vaandrager, F., Jansen, D.N., 2015. Applying automata learning to embedded control software. In: Butler, M., Conchon, S., Zaïdi, F. (Eds.), Formal Methods and Software Engineering. Springer International Publishing, Cham, pp. 67–83. http://dx.doi.org/10.1007/978-3-319-25423-4_5.

Snellman, N., Ashraf, A., Porres, I., 2011. Towards automatic performance and scalability testing of rich internet applications in the cloud. In: 2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications. pp. 161–169. http://dx.doi.org/10.1109/SEAA.2011.33.

Soldani, J., Tamburri, D., Heuvel, W., 2018. The pains and gains of microservices: A systematic grey literature review. J. Syst. Softw. 146, 215–232. http://dx.doi.org/10.1016/j.jss.2018.09.082, URL http://www.sciencedirect.com/science/article/pii/S0164121218302139.

Stankovic, N., 2006. Patterns and tools for performance testing. In: 2006 IEEE International Conference on Electro/Information Technology. pp. 152–157. http://dx.doi.org/10.1109/EIT.2006.252109.

Vögele, C., Hoorn, A., Schulz, E., Hasselbring, W., Krcmar, H., 2018. Wessbas: Extraction of probabilistic workload specifications for load testing and performance prediction–A model-driven approach for session-based application systems. Softw. Syst. Model. 17 (2), 443–477. http://dx.doi.org/10.1007/s10270-016-0566-5.

Wadler, P., 2012. Propositions as sessions. SIGPLAN Not. 47 (9), 273–286. http://dx.doi.org/10.1145/2398856.2364568.

Woodside, M., Franks, G., Petriu, D.C., 2007. The future of software performance engineering. In: 2007 Future of Software Engineering. FOSE '07, IEEE Computer Society, USA, pp. 171–187. http://dx.doi.org/10.1109/FOSE.2007.32.

Zhang, M., Ali, S., Yue, T., 2019. Uncertainty-wise test case generation and minimization for cyber-physical systems. J. Syst. Softw. 153, 1–21. http://dx.doi.org/10.1016/j.jss.2019.03.011.