The Journal of Systems and Software 237 (2026) 112832

Contents lists available at ScienceDirect

The Journal of Systems & Software

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

Intelligent automatic load test generation for elastic microservice applications: A falsification-based approach

Marco Zamponi [a,*], Daniele Masti [b], Emilio Incerto [a], Franco Raimondi [b], Mirco Tribastone [a]

[a] IMT School for Advanced Studies Lucca, Piazza San Francesco 19, 55100, Lucca, Italy
[b] Gran Sasso Science Institute, Viale F. Crispi 7, 67100, L'Aquila, Italy

ARTICLE INFO

Editor: Prof Raffaela Mirandola

Keywords:
Automated test generation
Model-based performance analysis
Quality assurance
Mixed integer optimization

ABSTRACT

Microservice applications are required to consistently guarantee Service-Level Agreements (SLAs) under fluctuating workloads, a challenge commonly addressed through autoscaling mechanisms. However, the effectiveness of an autoscaler strongly depends on the workload scenario, and validating robustness across diverse workload conditions remains an open problem. To address this, we propose an offline model-based framework that automatically generates load test traces designed to expose performance failures in elastic microservice applications. The system under test is modeled as a closed-loop dynamical system where the microservice application and the autoscaler are explicitly coupled. Specifically, we encode both components as piecewise affine functions, allowing a wide set of applications and autoscalers to be captured. Test generation is framed using a falsification approach and solved as a mixed-integer linear program, eliminating the need for manual configuration or real system interactions during test generation. The generated test cases are designed to cause SLA violations, uncovering critical workload scenarios that may be overlooked by existing approaches. We evaluate the framework on both a realistic benchmark microservice application and a population of randomly generated systems, demonstrating that the generated traces consistently induce performance failures in real deployments. Furthermore, we show that the method generalizes across different autoscaling policies and workload patterns, producing valid test traces within short time intervals. Finally, we discuss and compare alternative approaches for load test generation. These experiments highlight both the effectiveness of the approach in exposing performance violations and its applicability to diverse autoscaling configurations.

1. Introduction

A microservice architecture is typically defined as a distributed system composed of independently deployable services that communicate through lightweight protocols (Fowler, 2012). This architecture contrasts with traditional monolithic paradigms, which bundle all components into a single deployment unit (Dragoni et al., 2017; Velepucha and Flores, 2023). The success of the microservice approach stems largely from its support for independent development, testing, and deployment of services, which allows for quicker development cycles and facilitates parallel team workflows. These capabilities make microservices a preferred choice for large-scale applications designed to handle highly variable workloads; at the extremes, cloud applications may experience sudden flash crowds or sharp traffic spikes (Ari et al., 2003), conditions that can quickly deteriorate performance if not properly managed (Gunawi et al., 2016). Consequently, consistently guaranteeing desired quality of service remains challenging under these dynamic conditions. These requirements are often formalized through Service Level Agreements (SLAs) and defined as the degree to which the application meets functional and non-functional criteria (IEEE, 1990). Among these, performance is a critical concern as it heavily influences both user experience and business revenue (Deloitte Ireland, 2020).

A naive way to meet SLAs is to allocate a high number of resources to handle incoming workload. However, this incurs high costs, making the application economically unsustainable. For this reason, it is necessary to strike a balance between performance guarantees and deployment costs. The ability to dynamically acquire or release resources in response to changing workloads is commonly referred to as

* Corresponding author.
E-mail addresses: marco.zamponi@imtlucca.it (M. Zamponi), daniele.masti@gssi.it (D. Masti), emilio.incerto@imtlucca.it (E. Incerto), franco.raimondi@gssi.it (F. Raimondi), mirco.tribastone@imtlucca.it (M. Tribastone).

https://doi.org/10.1016/j.jss.2026.112832
Received 12 September 2025; Received in revised form 17 February 2026; Accepted 23 February 2026
Available online 25 February 2026

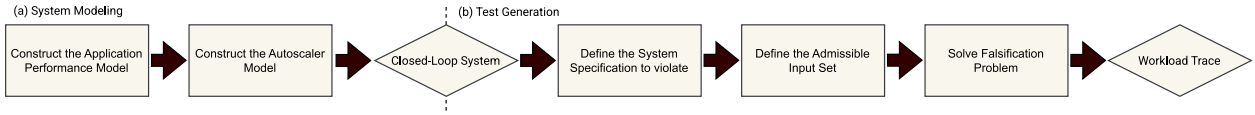

Fig. 1. Overview of the proposed automatic test generation methodology. (a) The dynamics of the microservice application under autoscaling are modeled as a PWA function over a finite time horizon. (b) Given the predictive model, a formal specification, and a defined test input space, test generation is framed as a falsification approach that is solved using MILP techniques.

elasticity (Herbst et al., 2013). To achieve elasticity, systems typically rely on a so-called *autoscaler*, a controller that continuously monitors the status of the application and determines the appropriate resource allocation (Lorido-Botran et al., 2014).

Determining suitable scaling actions from a vast set of possibilities is, however, a difficult task. To overcome this problem, the literature has explored diverse approaches. For example, rule-based approaches are employed due to their interpretability and ease of configuration (Lorido-Botran et al., 2014), but selecting effective rules remains difficult, and misconfigurations can cause performance degradation or resource waste (Dutreilh et al., 2010). Another popular approach is the one adopted by, e.g., Kubernetes Horizontal Pod Autoscaler (HPA) (Kubernetes, n.d.), a widely used tool in the context of cloud microservice applications, which varies resource allocations to adapt to performance targets but still requires careful parameter tuning (e.g., CPU utilization target, actuation range, sampling interval, stabilization window). Other sophisticated methods, such as the one based on mathematical optimization (Ghanbari et al., 2012; Megahed et al., 2017; Incerto et al., 2018, 2023), control theory (Baresi et al., 2016), or workload prediction (Iqbal et al., 2018; Golshani and Ashtiani, 2021; Zou et al., 2024), also offer great expressiveness but similarly require thorough validation and tuning to ensure reliability (Straesser et al., 2022). These issues are exacerbated by the fact that autoscaler efficacy heavily depends on the workload trace applied to the system: a policy that performs well under one trace may fail under another, depending also on the specific business objectives of the application (Papadopoulos et al., 2016). This highlights the need for systematic procedures to determine if a system governed by a particular autoscaler can consistently satisfy its performance requirements across a variety of workload traces. These requirements are often encoded using key performance indicators like under-provisioning time and magnitude, and their over-provisioning counterparts (Herbst et al., 2013). However, a systematic approach to this problem remains a largely unexplored area. In this setting, the main challenge stems from the coverage of the possible workload scenarios. This naturally leads to the question:

> *"How can one test whether an autoscaling policy is robust across the full spectrum of possible workload traces that a microservice application may face?"*

In this paper, we address this question by focusing on *load testing*, i.e., determining the load limits and shapes an elastic microservice application can tolerate on given hardware without violating performance requirements (Jiang and Hassan, 2015), an event we refer to as a *performance failure*. To do so, we propose an automatic load test generation approach to identify, from a family of valid workload traces, those most likely to trigger service degradation or failure-inducing conditions in a model-based fashion. In contrast to prior load test design studies, our methodology explicitly models the interaction between the microservice application and its autoscaler and frames the load test generation as an optimization problem. This enables the generation of compact, replayable workload traces *offline* (i.e., when the system is not running), thus avoiding manual configuration tuning and state-space exploration techniques that require costly experiments on the system.

Fig. 1 graphically summarizes the proposed approach. The first step, namely (a), involves constructing mathematical models of both the microservice application and the autoscaler. These models are then composed into a closed-loop representation of the system, which describes the evolution of performance metrics over time as a function of a workload trace. The closed-loop system is assumed to have a Piecewise Affine (PWA) structure with respect to the workload trace, enabling the encoding of diverse application and autoscaler models (Heemels et al., 2001).

In the second stage, namely (b), we introduce the *falsification problem*, defined as the search for an input that causes a system to violate its formal specifications (Mathesen et al., 2021). Specifically, we identify a workload trace that induces performance requirements violations. We encode the problem as a Mixed-Integer Linear Program (MILP), where the system PWA dynamics and the admissible workload traces are imposed as constraints. The optimization problem then generates a workload trace to minimize an objective defined in terms of the performance requirements.

We validate the proposed method through a three-fold evaluation: (i) by showing that generated workload traces cause performance failures in a real microservice benchmark and in a population of randomly generated systems; (ii) by testing its applicability across combinations of autoscaling policies, load patterns, and test objectives, discussing both test generation outcomes and computational costs (iii) by discussing alternative solutions and comparing them in terms of performance.

The paper is organized as follows. In §2 we review related work on cloud performance validation. §3 introduces the mathematical background of the paper. §4 presents a general framework to model microservice applications and autoscalers and proceeds to characterize the model closed-loop behavior. Subsequently, §5 presents the falsification problem and its formulation into an optimization problem that will be used to systematically generate workload traces aimed at identifying performance failures. We validate the proposed methodology in §6, demonstrating its ability to produce failure-inducing workload traces across different microservice applications as well as its capability in handling different autoscaler implementations and scenarios. The section concludes with a discussion of threats to validity. §7 discusses limitations of the approach, and §8 concludes the paper with directions for future work.

2. Related work

Existing approaches to cloud system load test design typically rely on statistical workload modeling from production traces (Calzarossa et al., 2016; Vögele et al., 2018). Orthogonally, in the direction of test optimization, in Vitui and Chen (2024) a machine-learning-based framework for early failing test stopping is presented. Another approach (Cooper et al., 2024) involves extracting short representative load tests from long historical workloads. While effective for validating expected system behavior under realistic usage scenarios, these techniques often fail to expose service-degrading conditions. Moreover, these approaches do not explicitly take into consideration the interplay between the system and the autoscaler. Other approaches focus on generating workload traces inducing performance issues, but either they operate online, making them difficult to reproduce and being slow to converge (Bayan and Cangussu, 2008), or focus on static system configurations, thus not allowing autoscaling dynamics testing (Barna et al., 2011). More recently, in the context of microservice applications, causal-reasoning based techniques have been proposed to generate static critical workloads (Giamattei et al., 2024; Mascia et al., 2025).

Beyond cloud performance testing load design, some works have specifically addressed the evaluation of autoscaling. For instance, Straesser et al. (2023) advocates for the importance of autoscaler performance evaluation with reasonable effort, and proposes autoscaler evaluation guidelines through qualitative anti-pattern identification over predefined load templates, albeit with limited and manual tuning guidelines. In parallel, Gambi et al. (2013) discusses a first conceptualization of load test design as a model-based optimization problem, further supporting the topic importance. Building on the need for systematic autoscaler evaluation approaches, we propose an automatic, model-based test generation method that explicitly accounts for the closed-loop interaction between microservice applications and autoscaling mechanisms. This allows the systematic exploration of both typical and adversarial workload traces that might be overlooked by statistical extraction methods. Each test is generated for a configurable fixed time horizon, thus the execution time is known in advance. Since our approach operates on the system model rather than through repeated online experimentation, it can freely explore the space of possible load tests, avoiding further costly measurements from the system under test. We note that the approach is not intended to replace tests derived from expected usage, but rather to complement them, in order to validate that the autoscaler behaves well even under adverse conditions.

Closely related to our approach, we mention formal verification approaches for elastic microservice applications (Papadopoulos et al., 2016; Evangelidis et al., 2018; Agos Jawaddi et al., 2024; Serracanta et al., 2025). Formal verification aims to prove that a model satisfies given properties through mathematical methods guaranteeing soundness. Our approach uses similar techniques, but with different goals: we employ mathematical programming not to prove correctness, but to construct workload traces that falsify requirements and expose performance failures. Several works (Evangelidis et al., 2018; Agos Jawaddi et al., 2024) extract probabilistic models from test traces of elastic applications, which are then analyzed through probabilistic model checking tools, such as PRISM (Hinton et al., 2006). In Papadopoulos et al. (2016) a probabilistic framework evaluates policies using chance-constrained programming to identify worst-case scenarios with respect to supply-demand curve distances, which are then used to compare different autoscaling techniques. Another study (Serracanta et al., 2025) provides a model for CPU-intensive microservice applications deployed through Kubernetes and formally proves the stability of the Kubernetes HPA.

3. Background

In this section we introduce the mathematical notation and queuing network models that will be used throughout the article.

Mathematical Notation. Scalars are denoted by lowercase letters (e.g., l, t), vectors by bold lowercase letters (e.g., $\mathbf{c}, \mathbf{s}$), and matrices by bold uppercase letters (e.g., $\mathbf{P}, \mathbf{Y}$). Overbars and underlines (e.g., $\bar{\mathbf{c}}, \underline{\mathbf{c}}$) denote, respectively, upper and lower bounds imposed on the corresponding quantities. Time-dependent quantities are written with explicit time indices (e.g., $l(t), \mathbf{c}(t)$). Symbol $\mathbf{I}_n$ represents the n-dimensional identity matrix, while $\mathbf{1}_n$ represents an n-dimensional vector of ones. Table 1 summarizes the notation and definitions of the symbols employed throughout the paper.

Queuing Network Models. To model the performance of resource-constrained systems, we resort to steady-state analysis of *closed* Queuing Networks (QNs) (Gordon and Newell, 1967), where a *fixed user population* l circulates in a network of n stations. Each station i is assigned a finite processing capacity. Specifically, the *resource allocation* vector $\mathbf{c}$ is defined as:

$$\mathbf{c} = [c_1, \dots, c_n]^\intercal, \quad \underline{\mathbf{c}} \le \mathbf{c} \le \bar{\mathbf{c}}.$$

Building upon fluid approximation theory (Bortolussi et al., 2013), the dynamics of such a QN can be described using a system of ordinary differential equations (ODEs), where each equation represents the time-course evolution of the *average queue length* $q_i(t)$ at service i:

$$\dot{q}_i(t) = -s_i(t) + \sum_{j=1}^{n} p_{j,i} s_j(t) \tag{1a}$$

$$s_i(t) = \mu_i \min(q_i(t), c_i), \tag{1b}$$

where $\mathbf{s}(t) = [s_1(t), \dots, s_n(t)]$ is the *throughput* vector, $\boldsymbol{\mu}$ the vector of exponential *service rates*, and $p_{j,i}$ the entries of the *routing probability matrix* $\mathbf{P}$, representing the probability of a user transitioning from service j to service i. Furthermore, the sum of users in each queue equals the total number of users in the network at every time step.

$$\sum_{i=1}^{n} q_i(t) = l \ \forall t. \tag{2}$$

We compactly represent the queue dynamics in Eq. (1a) in matrix form as $\dot{\mathbf{q}} = -\mathbf{I}_n \mathbf{s}(t) + \mathbf{P}^\intercal \mathbf{s}(t)$. At steady-state, the dynamics described by Eq. (1a) converge to an equilibrium point $\mathbf{q}^\infty(\mathbf{c}, l)$ that can be computed by solving the following flow balance equation:

$$\mathbf{P}^\intercal \mathbf{s}^\infty(\mathbf{c}, l) = \mathbf{s}^\infty(\mathbf{c}, l). \tag{3}$$

Here, the vector $\mathbf{s}^\infty(\mathbf{c}, l)$ represents the steady-state throughput of each service. Notably, due to the non-smooth $\min(\cdot)$ operator in Eq. (1b), the map $\mathbf{s}^\infty(\mathbf{c}, l)$ is piecewise affine (Kowal et al., 2015).

Consequently, we now formalize the percentage utilization $u_i(\mathbf{c}, l)$ and the response time $r_i(\mathbf{c}, l)$ at each service.

$$u_i(\mathbf{c}, l) = \frac{s_i^\infty(\mathbf{c}, l)}{\mu_i c_i} \quad r_i(\mathbf{c}, l) = \frac{q_i^\infty(\mathbf{c}, l)}{s_i^\infty(\mathbf{c}, l)}. \tag{4}$$

Table 1
Notation and definitions of the main symbols used in the paper.

Application Model (§4)	
n	Number of services (queues)
l	Active users
$\mathbf{c}$	Per-service CPU resource assignment vector
$\mathbf{q}$	Per-service average queue length vector
$\mathbf{s}$	Per-service throughput vector
$\boldsymbol{\mu}$	Per-service service rate vector
$\mathbf{P}$	Service routing probability matrix
$\mathbf{q}^\infty$	Steady-state average queue lengths
$\mathbf{s}^\infty$	Steady-state service throughputs
u	Service average utilization
r	Service average response time
l	Workload trace of active users over time
$\mathbf{y}$	Per-service autoscaler input vector
$\mathbf{Y}$	System evolution function
u^{up}	Upper utilization threshold of rule-based autoscaler
u^{low}	Lower utilization threshold of rule-based autoscaler
u^*	Target utilization for VP autoscaler implementation
Falsification Problem (§5)	
φ	Formal specification
ρ_φ	Robustness function of the formal specification
$\mathcal{L}$	Set of admissible workload traces
ε	Robustness function tolerance
r_{ref}	Reference station end-to-end average response time
τ	Under-provisioning response time violation limit
υ	Over-provisioning utilization violation limit
α, β	Trade-off parameters for robustness vs. load size

4. System modeling

As shown in Fig. 1a, the first step of the approach involves constructing a PWA encoding for both the microservice application and the autoscaler models. In this section we describe how these are derived.

4.1. Mathematical modeling of microservices

To capture the dynamics of the microservice application, we instantiate the closed QN formalism described in §3. In particular: (i) the network of n stations represents the set of deployed services; (ii) the resource allocation vector $\mathbf{c}(t)$ corresponds to the allocated CPU resources to each service; and (iii) the fixed population l corresponds to the number of active users circulating in the system.

This modeling formalism is particularly suitable for capturing the performance of CPU-intensive applications (Vilaplana et al., 2014). Despite their simplicity, queuing network models have been widely used in relevant elastic systems performance studies (Barna et al., 2011; Papadopoulos et al., 2016; Incerto et al., 2017, 2018; Gandhi et al., 2018; Tong et al., 2021; Quattrocchi et al., 2024).

To instantiate the model, one has to provide two sets of static parameters: the routing probability matrix $\mathbf{P}$ and the vector of service rates $\boldsymbol{\mu}$. These parameters can be inferred from standard system measurements (e.g., queue lengths, CPU utilization, response times) collected via monitoring tools using statistical resource demand estimation methods (Spinner et al., 2015). For instance, linear optimization methods can be employed to infer QN models directly from queue length measurements (Incerto et al., 2021).

Although closed QN models cannot capture many performance patterns caused by transient behavior, they still provide enough detail about the system dynamics to support load test generation.

4.2. Autoscaler model

We are now providing a PWA encoding of autoscalers. While the closed QN model has a fixed user population l and resource allocation $\mathbf{c}$, in the following, we assume that the closed network population within the system changes over time, i.e., $l(t)$ becomes an exogenous disturbance acting on the system. In this case, the autoscaler becomes crucial for rejecting the effect of workload variation by acting as a feedback controller modifying the resource allocation $\mathbf{c}(t)$ to maintain desired performance.

As mentioned in §1, the autoscaler periodically selects at each time step $t_k = t_0 + k\Delta_T$ an effective control action $\mathbf{c}(t_k)$, which will be maintained constant in the interval $[t_k, t_{k+1})$. In the following, we assume that the autoscaling interval Δ_T is large enough that, at each decision point t_k, the application has reached steady state with respect to the previously applied resource allocation $\mathbf{c}(t_{k-1})$. Likewise, the number of active users $l(t)$ is assumed to be almost constant in the interval $[t_{k-1}, t_k)$, i.e., $l(t) \approx l(t_{k-1})\ \forall t \in [t_{k-1}, t_k)$. This constant load approximation is common in related load testing studies (Herbst et al., 2015; Straesser et al., 2023; Cooper et al., 2024), and is easily implemented through standard load testing tools. Furthermore, this assumption also aligns with the behavior of production-grade autoscalers, which typically base decisions on metrics aggregated over several minutes (e.g., HPA Kubernetes, n.d., Amazon AWS with CloudWatch monitoring (Amazon Web Services, Inc., n.d.)).

The autoscaler bases its decisions on the steady state at time t_k, computed with respect to CPU assignments and active users sampled at t_{k-1}. Thus, to simplify notation, by Eq. (2) and Eq. (3) we define the steady state at time t_k as $\mathbf{s}^\infty(t_k) := \mathbf{s}^\infty(\mathbf{c}(t_{k-1}), l(t_{k-1}))$. Moreover, let $\mathbf{y}_i(t)$ be the vector of per-service inputs to the autoscaler PWA encoding, defined as follows:

$$\mathbf{y}_i(t_k) := [c_i(t_{k-1}), s_i^\infty(t_k)]^\intercal.$$

Consequently, following hybrid systems theory (Heemels et al., 2001), an autoscaler can be modeled as the following piecewise map:

$$c_i(t_k) = f_i(\mathbf{y}_i(t_k)) = \begin{cases} \mathbf{a}_1^\intercal \mathbf{y}_i(t_k) + b_1, & \text{if } \mathbf{y}_i(t_k) \in \mathcal{Y}_1 \\ \vdots & \vdots \\ \mathbf{a}_M^\intercal \mathbf{y}_i(t_k) + b_M, & \text{if } \mathbf{y}_i(t_k) \in \mathcal{Y}_M, \end{cases} \tag{5}$$

where sets $\mathcal{Y}_j \subset \mathbb{R}^2$ define a disjoint partition of the input-variable space, and parameters $\mathbf{a}_j \in \mathbb{R}^2$ and $b_j \in \mathbb{R}$ define the affine expression active in that region.

Many types of autoscaler implementations can be exactly or approximately represented using this structure, including rule-based controllers, step functions, and MPC controllers.

4.3. Autoscaler PWA representations

In this section, we present PWA formulations of the autoscalers that will be used in the experimental section.

Rule-Based Controller. A rule-based controller adjusts service CPU allocations based on utilization thresholds. For instance, if the current service utilization $u_i(\mathbf{c}(t_{k-1}), l(t_{k-1}))$ exceeds an upper threshold u^{up}, the controller increases the number of assigned cores. Conversely, if it falls below a lower threshold u^{low}, a specific number of cores is released. The controller policy can be represented as:

$$c_i(t_k) = \begin{cases} c_i(t_{k-1}) + 1, & \text{if } u_i(\mathbf{c}(t_{k-1}), l(t_{k-1})) \geq u^{\text{up}} \\ c_i(t_{k-1}) - 1, & \text{if } u_i(\mathbf{c}(t_{k-1}), l(t_{k-1})) \leq u^{\text{low}} \\ c_i(t_{k-1}), & \text{otherwise.} \end{cases} \tag{6}$$

Similarly, step functions can be encoded by adding additional branches to the PWA function, with the corresponding thresholds and core additions or removals.

Kubernetes HPA. The Kubernetes HPA controller maintains the average service utilization $u_i(\mathbf{c}(t_{k-1}), l(t_{k-1}))$ near a target value u^* by adjusting the number of assigned cores proportionally (Kubernetes, n.d.). In particular, the proportional update rule can be written as:

$$c_i(t_k) = c_i(t_{k-1}) \frac{u_i(\mathbf{c}(t_{k-1}), l(t_{k-1}))}{u^*}. \tag{7}$$

In this form, the equation involves multiplying two variables and thus does not fit within the PWA framework. However, substituting Eq. (4) into Eq. (7), we obtain the form

$$c_i(t_k) = c_i(t_{k-1}) \frac{s^\infty(t_k)}{\mu_i c_i(t_{k-1}) u^*} = \frac{s_i^\infty(t_k)}{\mu_i u^*}. \tag{8}$$

Assuming fixed μ_i and u^*, the formula is affine in $s_i^\infty(t_k)$ and can thus be exactly encoded as a PWA function. This update rule, without rounding, can yield fractional resources allocations, so Eq. (8) effectively encodes a *Vertical Proportional* (VP) autoscaler inspired by HPA. Further features of this autoscaler implementation (e.g., tolerances, limits, rounding, stabilization windows) can be encoded in the MLD framework (Bemporad and Morari, 1999) and are therefore admissible in the PWA framework adopted in this work (Heemels et al., 2001).

Model Predictive Control. Model Predictive Control is a family of control techniques where controllers solve an online optimization problem incorporating a predictive model of the system dynamics at each decision time step (García et al., 1989). This approach has found considerable use in autoscaling applications (Ghanbari et al., 2012; Incerto et al., 2018). When the underlying optimization problem is a multi-parametric convex program, the resulting controller can be explicitly represented as a PWA function of the system state (Bemporad et al., 2000a). As such, MPC controllers can be naturally included in the proposed framework using the formulation described in Eq. (5).

Remark 1 (Model Generality). Since the only requirement for our approach is the availability of a finite PWA representation of the system, the methodology extends beyond the modeling choices presented in §4.1 and §4.2.

For instance, regarding microservice applications, Layered Queuing Network (LQN) models enable the representation of multi-tier applications (Incerto et al., 2023), while tail-latency prediction models allow the verification of percentile requirements, which are common in industry (Herbst et al., 2015).

Regarding both complex application dynamics and autoscalers, we remark that ReLU-activated neural networks admit a finite PWA representation (Tjeng et al., 2017; Fabiani et al., 2025), thus can be directly integrated into the proposed approach. Furthermore, for black-box or complex components, piecewise regression techniques can be employed to construct PWA models with bounded complexity (Bemporad, 2023).

4.4. Closed loop modeling via hybrid system theory

Having defined the application and autoscaler models in PWA form, we compose them into a unified closed-loop representation within the mixed logical dynamical (MLD) framework (Bemporad and Morari, 1999). Unrolled over a horizon of T steps, the closed-loop evolution at times $\{t_0, t_0+\Delta_T, \dots, t_0+T\Delta_T\}$ is described by the conjunction of constraints derived from Eq. (2), Eq. (3), and Eq. (5), $\forall t = 1:T, \forall i = 1:n$ and thus can itself be expressed as a bounded PWA function of the initial resource assignment $\mathbf{c}(0)$ and the workload trace $\mathbf{l} = [l(1), \dots, l(T)]$ (Heemels et al., 2001). For compactness, we denote through a single function the mapping that returns the information of all services across all considered time steps. This is defined as

$$\mathbf{Y}(\mathbf{c}(0), \mathbf{l}) := \begin{bmatrix} \mathbf{y}(1)^\intercal \\ \mathbf{y}(2)^\intercal \\ \vdots \\ \mathbf{y}(T)^\intercal \end{bmatrix}, \quad \mathbf{y}(t) := \begin{bmatrix} \mathbf{y}_1(t) \\ \mathbf{y}_2(t) \\ \vdots \\ \mathbf{y}_n(t) \end{bmatrix}^\intercal. \tag{9}$$

For brevity, the dependence of $\mathbf{Y}$ on $\mathbf{c}(0)$ and $\mathbf{l}$ will sometimes be omitted.

Remark 2. Methodology Abstraction The approach presented in this paper is, in principle, applicable to any choice of workload, resources, and performance metrics, provided that a corresponding bounded PWA function relating load and resource inputs to performance metrics exists. While we instantiated the method using throughputs, CPU cores, and active user counts, alternative choices might include, for example:

Load: class mix ratios, request arrival rates;
Resources: memory, network bandwidth, GPU;
Performance metrics: latency quantiles, queue lengths, error rates.

5. Test generation as system falsification

As depicted in Fig. 1b, the core idea of our approach is to reframe the load test generation process using a *falsification* approach (Mathesen et al., 2021). This involves finding an input assignment that causes the closed-loop system to violate a given formal specification φ. We define the search procedure for such an input as the *falsification problem*. Inspired by cyber-physical systems literature (Bartocci et al., 2018), we employ a *robustness function* $\rho_\varphi(\mathbf{Y})$ to quantify the *degree of satisfaction* of a formal specification φ by a system *signal* $\mathbf{Y}$ (comprising throughputs and core allocations through time). This function is defined such that $\rho_\varphi(\mathbf{Y}) \leq 0$ represents a specification violation.

In our framework, the sole requirement for the chosen robustness function is the admission of a PWA representation (or an equivalent linear-integer encoding). This formulation is generic and can accommodate various specifications of interest. Indeed, φ can be formalized using logic frameworks such as *Signal Temporal Logic* (STL), which allows systematic encoding of complex requirements (e.g., instantaneous or sustained SLA compliance, recovery deadlines) into linear-integer constraints via its quantitative semantics (Bartocci et al., 2018; Lindemann and Dimarogonas, 2019).

To mitigate spurious violations (i.e., model violations not reproduced in the real system), thresholds $\epsilon \geq 0$, depending on the load test objective, can be incorporated in the falsification problem, thereby requiring stricter conditions for declaring a violation. In §5.2 we detail the robustness functions that will be used in the experiments.

The *falsification problem* thus corresponds to deciding the truth of the formula

$$\exists \mathbf{l} \in \mathcal{L} : \rho_\varphi(\mathbf{Y}) \leq -\epsilon, \tag{10}$$

where $\mathcal{L} \subseteq \mathbb{R}^T$ represents a set of admissible workload traces. We solve problem (10) using MILP techniques, which offer sufficient expressivity for encoding the PWA structure and efficient solver support.

5.1. Falsification problem MILP encoding

To solve the falsification problem (10), we formulate a MILP where the workload trace acts as a decision variable. The solver identifies an input sequence satisfying the negative robustness constraint while optimizing a user-defined cost. Formally:

$$\min_{\mathbf{l}} \quad J(\mathbf{Y}, \mathbf{l}) \tag{11a}$$
$$\text{s.t.} \quad (9) \tag{11b}$$
$$\mathbf{l} \in \mathcal{L} \tag{11c}$$
$$\rho_\varphi(\mathbf{Y}) \leq -\epsilon \tag{11d}$$

The objective value function $J(\cdot)$ in (11a), defined over the output trajectory $\mathbf{Y}$ and the load trace $\mathbf{l}$, is used to encode the test generation goal. In this work, given a specification φ and its corresponding robustness function ρ_φ, we consider objective functions of the form

$$J(\mathbf{Y}, \mathbf{l}) = \alpha \rho_\varphi(\mathbf{Y}) + \beta \left(\sum_{t=2}^{T} |l(t) - l(t-1)| + \sum_{t=1}^{T} l(t) \right), \tag{12}$$

where the first term is used to minimize the robustness of the specification (thus inducing performance failures), while the second term penalizes load variations and high active user counts. Parameters $\alpha, \beta \in \mathbb{R}^+$ are weights used to balance these terms. Constraint (11b) acts as a shortcut to compactly encode the dynamics of the closed-loop system. Particularly, the constraints model the system state evolution as a sequence of steady-states reached by the application under the influence of autoscaler and disturbance actions, making our approach fundamentally different from works such as Bemporad et al. (2000a,b), which explicitly focus on transient behavior analysis. Constraint (11c) restricts the search to workload traces within the admissible set (detailed in §5.3), while Constraint (11d) enforces violation of the target specification. Since objectives and constraints of the optimization problem (11) are expressed through logical or linear inequalities, it can be solved using off-the-shelf MILP solvers.

Computational Complexity. For a finite time horizon T, the MILP encoding of the n-station closed queuing network throughputs from Eq. (3) requires nT binary variables to represent the $\min(\cdot)$ resource saturation operators at each time step. Furthermore, encoding the autoscaler policy Eq. (5) with M regions requires nMT additional binary variables to encode the active PWA region for each station at each time step.

5.2. System specification encodings

We hereby provide specifications accounting for resource under-provisioning (Herbst et al., 2015) and over-provisioning (Lorido-Botran et al., 2014). In these specification formulations, the robustness function quantifies the total deviation from ideal behavior. A system failure is declared only if the accumulated magnitude of violations over the time horizon exceeds the tolerance (i.e., $\rho_\varphi \leq -\epsilon$), which effectively identifies sustained performance degradation.

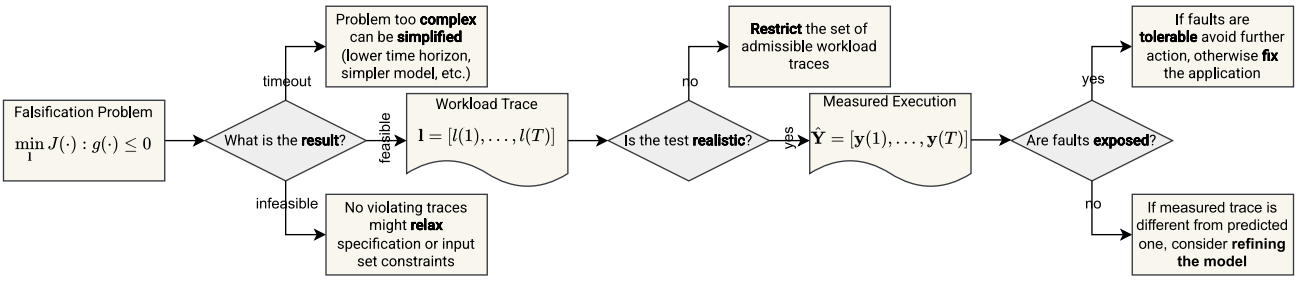

Fig. 2. Software engineer workflow for evaluation and refinement of a generated test case. The process begins with the solution of the optimization problem (11). Based on the outcome, the engineer may revise the test generation details or accept the test. The test is then replayed on the real system, where performance metrics are measured and compared against predicted ones to determine whether model refinement or application corrective actions are needed.

Average Response Time Specification. As a proxy for under-provisioning situations, we consider a performance specification that measures the occurrence of end-to-end average response time (i.e., the average time required to traverse the network, excluding the user think time) exceeding τ seconds. The end-to-end average response time is computed as

$$r_{\text{ref}}(\mathbf{c}, l) = \frac{l}{s^{\infty}_{\text{ref}}(\mathbf{c}, l)} - \frac{1}{\mu_{\text{ref}}}. \tag{13}$$

A violation occurs when $r_{\text{ref}}(\mathbf{c}, l) \geq \tau$. In order to allow encoding the specification in the PWA framework, considering Eq. (13) and the fact that the reference station, mimicking the user think time, is always unsaturated by definition (i.e., $s^{\infty}_{\text{ref}}(\mathbf{c}, l) = \mu_{\text{ref}} q^{\infty}_{\text{ref}}(\mathbf{c}, l)$), we rewrite this condition as

$$l - (\tau\mu_{\text{ref}} + 1) q^{\infty}_{\text{ref}}(\mathbf{c}, l) \geq 0. \tag{14}$$

The left-hand side of constraint (14) allows us to quantify the violation given our technical constraints. To measure the magnitude of violations while ignoring non-violating states, we introduce the Response Time Violation (RTV) metric:

$$RTV(\mathbf{c}, l) = \max\left(l - (\tau\mu_{\text{ref}} + 1) q^{\infty}_{\text{ref}}(\mathbf{c}, l), 0\right).$$

Let φ be a specification requiring zero cumulative response time violation over the time horizon T. Consequently, we define the robustness function ρ_{φ} providing quantitative semantics for the specification as

$$\rho_{\varphi}(\mathbf{Y}) = -\sum_{t=1}^{T} RTV(\mathbf{c}(t), l(t)). \tag{15}$$

Cores Over-provisioning. We consider a performance specification that limits resource over-provisioning over the time horizon. Specifically, we target scenarios where the CPU utilization of a service falls below $\upsilon \in [0, 1)$, corresponding to $u_i(\mathbf{c}, l) \leq \upsilon$. Since the utilization definition (see Eq. (4)) involves a division between variables, this condition cannot be exactly encoded in the PWA framework; thus we rewrite it as $s^{\infty}_i(\mathbf{c}, l) \leq \upsilon\mu_i c_i$. Considering the throughput definition Eq. (1b), if a station is saturated (i.e., $s^{\infty}_i(\mathbf{c}, l) = \mu_i c_i$), the over-provisioning condition is never satisfied; thus we consider $s^{\infty}_i(\mathbf{c}, l) = \mu_i q^{\infty}_i(\mathbf{c}, l)$. The condition can then be rewritten as

$$\upsilon\mu_i c_i - \mu_i q^{\infty}_i(\mathbf{c}, l) = \upsilon c_i - q^{\infty}_i(\mathbf{c}, l) \geq 0. \tag{16}$$

The left-hand side of Constraint (16) provides a measure for the specification violation. In particular, we define the over-provisioning magnitude metric as

$$o_i(\mathbf{c}, l) = \max(\upsilon c_i - q^{\infty}_i(\mathbf{c}, l), 0).$$

In turn, we introduce a specification φ requiring zero cumulative resource over-provisioning across all time steps and services, encoded through the following robustness function:

$$\rho_{\varphi}(\mathbf{Y}) = -\sum_{t=1}^{T}\sum_{i=1}^{n} o_i(\mathbf{c}(t), l(t)). \tag{17}$$

5.3. Admissible workload traces

The nature of the set $\mathcal{L}$ is a central issue in our proposal. The first, simplest option would be to set $\mathcal{L} = \{\mathbf{l} : \underline{l} \leq l(t) \leq \bar{l}, \forall t = 1{:}T\}$, i.e., impose no constraints on the workload shape. In this case, the optimization problem (11) seeks the worst-case scenario for the autoscaler, i.e., the workload trace inducing the highest performance violation. However, this approach might produce highly unrealistic test traces that would then require further refinement steps, and it makes solving problem (11) unnecessarily complex (see, e.g., Ágoston and Marianna, 2024). We adopt a slightly stricter approach, referred to as *Free Load Shape*, which includes a short initial warm-up phase and imposes bounds on the rate of change of the active user count between successive time steps, preventing unrealistic abrupt variations. The corresponding set $\mathcal{L}$ can be formalized as the conjunction of the following constraints.

$$\begin{aligned} \underline{l} \leq l(t) \leq \bar{l} \quad & \forall t = 1{:}T \\ l(t) = l(t-1) \quad & \forall t = 2{:}\tau_1 \\ |l(t) - l(t-1)| \leq \sigma \quad & \forall t = \tau_1{+}1{:}T, \end{aligned} \tag{18}$$

with $\tau_1 = \lfloor 0.15T \rfloor$ being the final time step of the warm-up period and σ denoting a scalar value that bounds the maximum rate of change.

Following cognate literature (Straesser et al., 2023; Gunawi et al., 2016; Ari et al., 2003), we therefore explore restricting $\mathcal{L}$ to canonical load shapes, similarly defined as the one above, such as:

- *Ramp Load Shape*: It consists of an initial flat warm-up period, a linear increase (or decrease) phase, and a steady phase. This pattern is used to expose situations where the autoscaler reacts either too slowly or too harshly to the workload change (Straesser et al., 2023).
- *Spike Load Shape*: Sudden workload bursts are a common cause of microservice failures (Gunawi et al., 2016). We model load spikes as an initial flat warm-up period followed by a sudden load increase and then an exponential decay.
- *Sawtooth Load Shape*: Real workloads are often characterized by oscillating behavior. The sawtooth shape aims to approximate these periodic patterns.

For brevity, the detailed encodings of the load shape constraints are omitted here and deferred to Appendix A. In general, admissible load definitions inherently depend on the system objectives, and the proposed framework is expressive enough to encode a wide variety of application-specific requirements. For example, additional constraints can be introduced to limit abrupt changes in user count, or to enforce similarity with empirical traces extracted from production logs, thus finally resulting in representative workload traces.

5.4. Test evaluation workflow

We discuss an end-to-end workflow that could be followed by a practitioner for the evaluation of a generated test case and suggested actions based on the outcomes. The workflow is graphically depicted in Fig. 2.

We begin by considering the possible outcomes of the solution of the optimization problem (11). First, we rule out the problematic cases where the solver fails to terminate within the allotted time limit, neither returning a feasible solution nor proving infeasibility. In such a case, nothing can be concluded. If instead problem (11) has been found to be infeasible, no falsifying traces can be found. In this case, a practitioner may relax certain optimization constraints to generate alternative workload traces for testing the system.

If the solver returns a set of falsifying workload traces, the practitioner can inspect them to decide whether they are realistic and, if so, use them to validate the real system. The execution of the test on the real system will lead to a vector of measured values $\hat{\mathbf{Y}}$. If the test indeed exposes the predicted performance failures, the practitioner can decide whether the observed behavior can be tolerated or if the autoscaler needs to be refined. If, instead, the measured performance does not reflect the predicted one, the generated test can be used to identify the conditions in which the model fails and can serve as a basis to refine it.

Finally, if a generated load trace is considered unrealistic, specifications (such as the nature of $\mathcal{L}$) can be restricted and a new test generation process launched.

6. Results

This section presents an evaluation of the proposed model-based framework for load test trace generation. In particular, we validate the proposed approach by answering the following questions:

- **Q1:** Is our technique effective in generating load traces that can cause failures in a real microservice system?
- **Q2:** How does the proposed test generation approach perform across combinations of diverse autoscaling policies, load patterns, and test objectives?
- **Q3:** How does the proposed test generation approach compare with alternative adversarial load testing approaches?

To address **Q1**, we apply the test generation framework described in Fig. 2 to a realistic elastic microservice application, namely AcmeAir (Incerto et al., 2023, 2025). To verify that failures are present in the real system, we compare the performance measures predicted during test synthesis against the empirical measurements collected during test execution. Furthermore, we investigate the framework robustness to model inaccuracies by testing whether workload traces derived from perturbed models consistently expose system failures. Finally, to generalize beyond a single system, we extend the analysis to a population of randomly generated microservice applications, verifying the consistency of the validity of generated tests. This establishes the practical **effectiveness** of the proposed approach.

To address **Q2**, we use our technique to generate a set of workload traces under different autoscaling policies and across a variety of scenarios. We analyze the test generation outcomes and solving times to evaluate practical usability. This establishes the practical **applicability** of the proposed approach.

Finally, to address **Q3**, first we examine approaches that search for falsifying traces directly on the system under test. Since online experiments on the system under test are prohibitively costly, we reproduce on the model a naive random search and a genetic algorithm. From this, we compute the amount of samples needed to obtain a violation and draw conclusions on their applicability on real systems. Second, we compare two model-based search procedures for the falsification problem: the proposed MILP encoding and the genetic algorithm. We compare their performance in terms of time-to-violation and discuss further implications of the solving methodology choice.

All experiments were executed on an AWS `c6g.12xlarge` instance equipped with 48 vCPUs and 64 GiB of memory. The solutions of the optimization problem (11) were computed using Gurobi 12.0.1 as the MILP solver (Gurobi Optimization, 2024). The source code of the experiments and the produced data can be found in the replication package[1].

6.1. Microservice application description

The microservice applications used in the experiments consist of a group of stateless services, each deployed in its own *Docker* container. External HTTP requests are routed to the microservices via a *Traefik* reverse proxy instance through endpoint-based routing. Each service exposes a *Flask*-based HTTP endpoint emulating CPU load for a task duration sampled from an exponential distribution. This experimental design choice, already used in related works (Incerto et al., 2023, 2025), isolates CPU saturation dynamics from implementation-specific noise that would affect the methodology evaluation. Containers run multiple parallel processes equal to the maximum number of assignable CPU cores. These processes share the assigned CPU resources following a processor-sharing (PS) scheduling policy, managed by the underlying OS scheduler (Pabla, 2009).

Performance metrics are collected by a *Prometheus* instance from two sources: (i) *cAdvisor*, providing container-level CPU utilization; (ii) *Traefik*, which exposes application-level metrics, i.e., HTTP request rates and response times. The autoscaler component periodically queries aggregate performance metrics from *Prometheus*, evaluates the scaling policy, and performs vertical scaling by enforcing the computed CPU quotas via the *Docker API*. In our experiments, the autoscaler executes a control action every 3 time intervals of Δ_T seconds.[2]

6.2. Test execution

In the following experiments, we perform a load test on the target applications using the *Locust* workload generator. Given a generated load trace $\mathbf{l} = [l(1), \dots, l(T)]$, the *Locust* runner adapts the user count every Δ_T seconds, simulating user traffic. Each *Locust* user waits for an exponentially distributed amount of time (i.e., *think time*), selects the first station to visit according to the probability distribution $\mathbf{p}_e$, and then selects successive stations through the routing matrix $\mathbf{P}$.

6.3. *Q1: Effectiveness and robustness evaluation*

This section addresses Q1 by evaluating whether the workload traces generated by our model-based approach effectively expose performance failures in real-world deployments.

6.3.1. AcmeAir

In this experiment, we generate a test on a microservice application faithfully replicating the topology and resource demands of AcmeAir, a commonly used benchmark for studying performance-related aspects of microservice systems (Incerto et al., 2023, 2025). The application consists of 9 endpoints, which are individually deployed in their corresponding containers as described in §6.1. In this experiment, to facilitate load testing, we consider that the average service time of each service is five times the values of the layered queuing network model derived in Incerto et al. (2023), thus assuming the following values in seconds:

$$\hat{\mathbf{r}} = \begin{bmatrix} 0.5185 & 0.5075 & 0.363 & 0.159 & 0.284 \\ 0.1375 & 0.623 & 0.2615 & 0.464 & \end{bmatrix}.$$

To apply our methodology, we model the application behavior through a QN composed of $n = 10$ stations, where the first one is a reference station, while each of the remaining ones corresponds to a specific user-accessible endpoint. To stress all services, each user initially selects one of six services with equal probability and then traverses the system

[1] https://doi.org/10.5281/zenodo.17938632

[2] Vertical scaling is adopted here as it enables experiments to be conducted with limited assignable CPU cores.

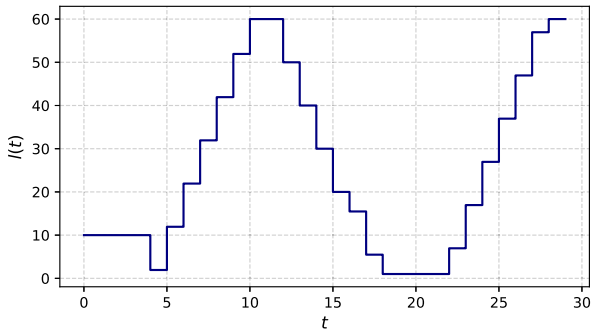

Fig. 3. Load test trace for the AcmeAir benchmark under the VP autoscaler inducing under-provisioning failures. Active user count at each time step t (where $\Delta_T = 60s$).

according to the inter-service communications of the microservice application. The resulting transition matrix is defined as

$$\mathbf{P} = \begin{bmatrix} 0 & 1/6 & 1/6 & 1/6 & 0 & 1/6 & 0 & 1/6 & 0 & 1/6 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 2/3 & 0 & 0 & 0 & 1/3 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

The service rates of the stations are the reciprocals of the average service times $\hat{\mathbf{r}}$. Thus, service rates and per-service core allocation bounds are

$$\boldsymbol{\mu} = [1, 1/\hat{r}_1, \ldots, 1/\hat{r}_9] \quad \underline{\mathbf{c}} = [\bar{l}, \underbrace{1, \ldots, 1}_{9 \text{ times}}] \quad \bar{\mathbf{c}} = [\bar{l}, \underbrace{4, \ldots, 4}_{9 \text{ times}}].$$

Experimental Setup. In this experiment, we generated a workload trace over a time horizon of $T = 30$ time steps, considering a maximum number of users circulating in the network $\bar{l} = 60$. The trace was constructed to violate the **under-provisioning** specification whose robustness function is defined in Eq. (15), with parameters $\tau = 1$ and $\varepsilon = 20$ (i.e., the specification is violated as long as the cumulative RTV is greater than 20). The application scaling actions are governed by the **VP** autoscaler implementation described in Eq. (8) with $u^* = 0.5$. We used the **free load shape** setting (with $\sigma = 10$, $l(0) = 10$). To generate the trace, we formulated and solved the corresponding optimization problem (11), imposing a time limit of 600 seconds on the solver. A feasible workload trace was found and is graphically represented in Fig. 3. In particular, the figure shows the number of active users at each time step. With no load shape imposed, the optimizer produced an oscillatory trajectory that represents a highly challenging workload for the chosen autoscaler.

The generated workload trace is then replayed on the real microservice application according to the procedure described in §6.2, with a time interval of $\Delta_T = 60$ seconds. For each test, we collected average throughputs and response times for each service across all time steps. Finally, the average end-to-end response time $r_{\text{ref}}(t)$ is computed from the average response times of the individual services. To mitigate the effect of inherent performance variability in cloud environments, the load test is repeated over 30 statistically independent runs.

Results. Fig. 4 shows the throughputs under the generated workload trace; dashed lines denote measured throughputs with shaded areas representing the corresponding standard deviation, while solid lines indicate predicted throughputs. Fig. 5, instead, reports the user end-to-end average response times $r_{\text{ref}}(t)$; dashed lines show measured response

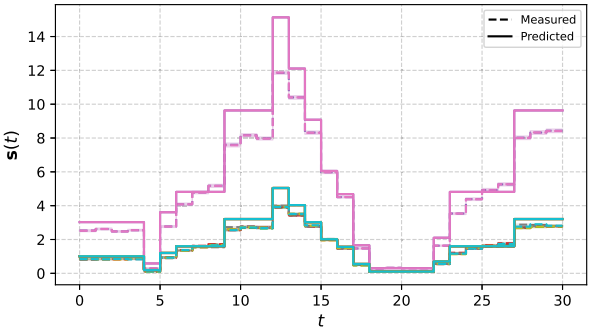

Fig. 4. Request throughputs for each service at each time step t (where $\Delta_T = 60s$). Dashed lines (with shaded standard deviation) show measured values; solid lines denote model predictions. Distinct colors correspond to different services.

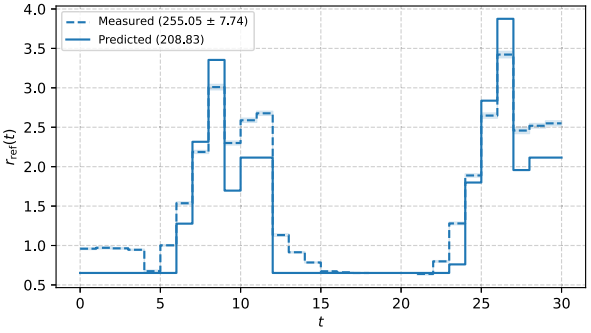

Fig. 5. Average end-to-end response times at each time step t (where $\Delta_T = 60s$). Dashed lines (reconstructed from service-level measurements, with shaded standard deviation) show measured values; solid lines show model predictions. Legend values in parentheses represent the measured (with standard deviation) and predicted cumulative RTV metric.

times, with shadings representing the standard deviation across test repetitions, while solid lines represent values predicted by the performance model.

The results of these experiments are two-fold: (i) generated workload traces indeed expose performance failures in the real system, and (ii) this effect persists even in cases of model mismatch.

To rigorously support this observation, we performed a one-sample Wilcoxon signed-rank test[3] (Wilcoxon, 1945) on the violation margins $D = |\rho_\varphi| - \varepsilon$. We tested the null hypothesis $H_0 : \tilde{m}_D \leq 0$ (median violation is within tolerance) against the alternative $H_1 : \tilde{m}_D > 0$ (median violation exceeds tolerance), at a significance level $\alpha = 0.01$. The test yielded a p-value $\approx 9.31 \times 10^{-10} < \alpha$, thus rejecting the null hypothesis and supporting the claim that the generated workload trace effectively exposes failures in the system under test.

The predictive model closely matches the measured throughputs except during intervals approaching resource saturation (e.g., time steps 9–15 and 27–30). In these cases, measured throughputs are smaller than the predicted ones. This phenomenon is likely caused by the fact that queuing network models tend to overestimate the performance capabilities of systems close to resource saturation (Malone et al., 2007). Consequently, we can also see how the model tends to underestimate the application response time, which makes the test generation procedure conservative. Indeed, in the case of the under-provisioning specification, the real system is likely to experience worse performance than the predicted one given a generated workload trace, reducing the risk of false

[3] We used the non-parametric Wilcoxon signed-rank test since the normality of the *Cumulative RTV* cannot be assumed.

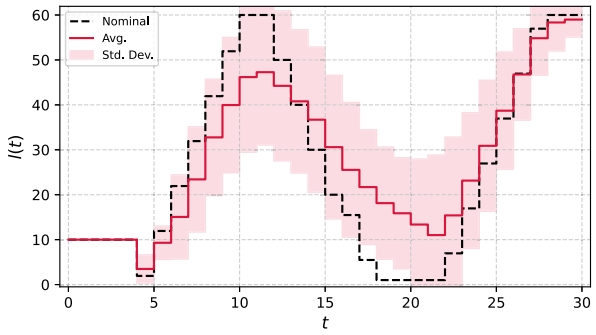

Fig. 6. Average workload trace across 30 perturbed AcmeAir models (solid, shaded standard deviation) compared with the base model trace (dashed), at each time step t (where $\Delta_T = 60s$).

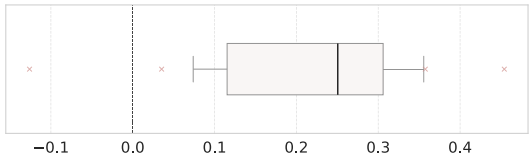

Fig. 7. Relative error between measured and predicted cumulative violation magnitudes (difference normalized by the measured value): positive values indicate model underestimation; negative values indicate overestimation. Plot shows median and quartiles (box), 5th-95th percentiles (whiskers), and outliers (crosses).

positive workload traces. We further explore this hypothesis in the next experiment.

6.3.2. Sensitivity to model mismatch

To further assess the practical applicability of the framework, we investigate the sensitivity of the test generation process against model parameter estimation errors. We conducted a sensitivity analysis by introducing random perturbations to the service rates $\boldsymbol{\mu}$ of the base AcmeAir model described in §6.3.1. We generated a set of $N = 30$ perturbed models where each service rate vector $\hat{\boldsymbol{\mu}}^j$ is distorted by adding uniformly distributed parametric uncertainty $\boldsymbol{\delta}^j$, with $\delta_i^j \sim \mathcal{U}(-0.2, 0.2)\mu_i, \forall i = 1:n$, representing a ±20% uncertainty margin relative to the base model values, $\forall j = 1:N$.

Following the experimental procedure detailed in §6.3.1, we solved the optimization problem (11) for each model instance j, $\forall j = 1:N$ with corresponding service rates $\hat{\boldsymbol{\mu}}^j$ and generated a workload trace $\mathbf{l}_j$. The resulting workload traces are then replayed on the unchanged real microservice application. We recorded average service throughputs and response times across time, and consequently computed average end-to-end response times $r_{\text{ref}}(t)$.

Results. Fig. 6 compares the average workload trace derived from the perturbed models against the base model trace obtained in the previous experiment. Notably, the shape of the average load trace generated on perturbed models resembles the one generated from the base model.

All measured cumulative RTV metrics exceeded the target threshold $\varepsilon = 20$, with a median of 250.80. To validate the significance of these violations, we applied the same one-sample Wilcoxon signed-rank test procedure presented in §6.3.1. The test yielded a p-value $\approx 9.31 \times 10^{-10} < \alpha$, thus rejecting the null hypothesis and demonstrating that generated workload traces expose failures despite model uncertainty.

6.3.3. Random systems

We further support **Q1** and assess the statistical and generalization properties of our approach by performing a Monte Carlo study on $N = 50$ different microservice applications, each composed of 3 services with different service rates, obtained by sampling uniform distributions. In particular, $\mu_i \sim \mathcal{U}(1, 10), \forall i = 1:n$. The user think rate is also sampled from a uniform distribution $\mu_{\text{ref}} \sim \mathcal{U}(0.5, 2)$. Thus, final service rates and per-service core allocation bounds are

$$\boldsymbol{\mu} = [\mu_{\text{ref}}, \mu_1, \mu_2, \mu_3] \quad \underline{\mathbf{c}} = [\bar{l}, 1, 1, 1] \quad \bar{\mathbf{c}} = [\bar{l}, 8, 8, 8].$$

Next, we construct a strongly connected stochastic matrix P describing the probabilities of moving from one station to the next. We performed rejection sampling, excluding all random instances whose expected r_{ref} in an unsaturated regime exceeds 0.5 seconds.

For each application we then generate, through the solution of the optimization problem (11), a workload trace for a time horizon of $T = 24$ time steps, with the **sawtooth** load shape for the VP autoscaler Eq. (8) with $u^* = 0.5$ and targeting the application under-provisioning specification Eq. (15) (with $\tau = 1$ and $\varepsilon = 20$). Subsequently, in cases where a falsifying trace for the system was found, we also replay the load test as described in §6.2, with a time interval $\Delta_T = 20$ seconds, measuring the average throughputs and response times over each time fragment.

Results. For each randomly generated microservice application, the corresponding optimization problem terminated within the defined time limit, thus yielding a conclusive result in all cases. In particular, among all 50 generated instances, 23 admit a falsifying workload trace for which the optimal solution was identified; the remaining instances were ultimately infeasible, thus confirming that no falsifying traces exist in the model.

We computed the cumulative RTV metric from the simulations of each of the 23 systems for which we found a feasible falsifying load trace. We applied the one-sample Wilcoxon signed-rank test procedure described in §6.3.1 on the measured cumulative RTV metrics of the 23 experiments against the threshold $\varepsilon = 20$. The test yielded a p-value $\approx 1.19 \times 10^{-7} < \alpha$, thus rejecting the null hypothesis and supporting the claim that the approach is successful in generating failure-inducing workload traces across the randomly generated system configurations.

To confirm the hypothesis in §6.3.1, which states that measured system performance is worse than that predicted by the model in the under-provisioning case, we computed the relative error between the measured and predicted cumulative RTV values for each test defined as $(|\rho_\varphi(\hat{\mathbf{Y}})| - |\rho_\varphi(\mathbf{Y})|)/|\rho_\varphi(\hat{\mathbf{Y}})|$, with $\hat{\mathbf{Y}}$ being the measured execution trace and $\mathbf{Y}$ the predicted one. Fig. 7 shows this metric as a box plot and indicates that the model underestimates violations in most cases.

To statistically validate this hypothesis, we performed a paired Wilcoxon signed-rank test comparing measured and predicted violation magnitudes. Specifically, we applied the test on the paired differences $D = |\rho_\varphi(\hat{\mathbf{Y}})| - |\rho_\varphi(\mathbf{Y})|)$ for the feasible instances. We tested the null hypothesis $H_0 : \tilde{m}_D \leq 0$ (the median measured violation does not exceed the prediction) against the alternative $H_1 : \tilde{m}_D > 0$ (the median measured violation exceeds the prediction), at a significance level $\alpha = 0.01$. The test yielded a p-value $\approx 8.34 \times 10^{-7} < \alpha$, thus rejecting the null hypothesis and supporting the claim that the real system violation magnitudes exceed the predicted ones, and validating the use of QN models as a conservative proxy for under-provisioning failures.

Answer to Q1: The proposed approach effectively generates workload traces that consistently expose performance failures in the real microservice applications. Further experiments also demonstrate that the methodology is robust to model mismatch and generalizes effectively across diverse system configurations.

6.4. *Q2: Test suite generation*

Having demonstrated the capability of the methodology in producing failure-inducing tests, in this experiment we address **Q2** by performing

Table 2
Summary of test generation outcomes for over-provisioning (top) and under-provisioning (bottom) across all combinations of autoscalers and load shapes. Cells marked (✓) report a falsifying trace, and performance is reported as (Time to first feasible load / Time to optimal load), where *lim* indicates a timeout. Cells marked (✗) indicate problem infeasibility, and performance is reported as (Time to prove infeasibility).

Over-provisioning objective				
Autoscaler	Free	Ramp	Sawtooth	Spike
VP50	✓(0.07/11.41)	✓(0.53/2.07)	✓(0.40/4.94)	✓(6.30/13.34)
VP60	✓(0.21/4.78)	✓(0.12/4.27)	✓(0.40/7.06)	✓(0.71/6.83)
Step1	✓(0.17/77.85)	✓(0.07/3.60)	✓(0.15/13.53)	✓(0.13/5.80)
Step2	✓(0.14/lim)	✓(0.13/27.18)	✓(0.67/lim)	✓(17.44/280.47)
Under-provisioning objective				
Autoscaler	Free	Ramp	Sawtooth	Spike
VP50	✓(0.30/lim)	✗(0.50)	✗(2.18)	✓(2.64/2.96)
VP60	✓(0.32/lim)	✗(2.79)	✓(0.19/7.54)	✓(0.45/3.28)
Step1	✓(0.20/10.64)	✓(0.09/1.15)	✓(2.18/2.79)	✓(0.53/0.97)
Step2	✓(1.57/326.78)	✗(2.52)	✓(11.51/13.69)	✓(3.01/4.32)

a feasibility assessment of the test generation procedure. To do so, we generate failure-inducing workload traces across different autoscaling policies and scenarios (i.e., combinations of load shapes and test objectives). Consequently, we analyze the outcomes of the optimization problems and the corresponding computational costs.

Concerning the system, we consider a two-tier application composed of a lightweight front-end and two slower back-end services. The system is modeled as a closed QN with $n = 4$ stations, the first one being a reference station, while the other three represent CPU-bound services. User requests circulate among services according to the routing matrix

$$\mathbf{P} = \begin{bmatrix} 0 & 1.0 & 0 & 0 \\ 0 & 0 & 0.6 & 0.4 \\ 0.7 & 0.3 & 0 & 0 \\ 0.8 & 0.2 & 0 & 0 \end{bmatrix}.$$

Service rates and per-service core allocation bounds are defined as

$$\boldsymbol{\mu} = [1, 8, 4, 4] \quad \underline{\mathbf{c}} = [\bar{l}, 1, 1, 1] \quad \bar{\mathbf{c}} = [\bar{l}, 8, 8, 8].$$

Experimental Setup. We generate $T = 24$ time-step-long tests designed to expose different types of performance failures under various load conditions. The maximum number of concurrent users is fixed to $\bar{l} = 80$. Autoscaling decisions are evaluated every 3 time steps based on the average CPU utilization observed over the previous 3 time steps. For this experiment, we considered all the combinations of the following components for the optimization problem (11):

Candidate Autoscalers. In this experiment, we consider the following autoscaling policies: (**VP50**) VP implementation Eq. (8) with a target utilization $u^* = 0.5$, (**VP60**) with $u^* = 0.6$, (**Step1**) a step controller Eq. (6) with $u^{\text{low}} = 0.3, u^{\text{up}} = 0.7$, and (**Step2**) a multi-step controller with thresholds $[0.2, 0.4, 0.6, 0.8]$ and corresponding scaling actions $[-2, -1, 0, 1, 2]$. The autoscaler implementation is integrated into the optimization problem as part of the equations characterizing constraint (11b).

Load Shapes. Here we focus on the admissible workload sets described in Section 5.3. In particular, we consider (i) **free** load shape, (ii) **ramp** load shape, (iii) **spike** shape, and (iv) **sawtooth** pattern. All the constraints associated with the selected load shape are added to the falsification problem through constraint (11c).

Test Objectives. In this experiment, we consider the following specifications: (i) **under-provisioning** (Eq. (15) with $\tau = 1$ and $\varepsilon = 20$), and (ii) **over-provisioning** (Eq. (17) with $\upsilon = 1/2$ and $\varepsilon = 20$). Each goal is represented through its robustness function, which is minimized in the optimization problem alongside penalties on the norm and variation of user loads over the time horizon, as described in Eq. (12). Moreover, they are used in constraint (11d), enforcing the generation of tests falsifying the specification.

By considering all the possible combinations of the described components (4 autoscalers × 4 load shapes × 2 test objectives), we solved a total of 32 test generation optimization problems. We set a time limit of 10 minutes per instance on the solver. If a feasible solution was found within the time limit, it was accepted even if suboptimal.

Results. Table 2 summarizes the outcome of the test generation problems across all combinations of autoscalers, load shapes, and test objectives. Results are presented in two tables, collecting the data from the test generation for the over-provisioning (top) and under-provisioning (bottom) objectives. Each cell indicates whether the falsification problem was solved successfully (i.e., a falsifying workload was found), marked with a tick, or proven to be infeasible (i.e., no falsifying workload exists under the imposed constraints), marked with a cross; for feasible cases (✓), the accompanying numbers report the total solving time in seconds to the first feasible and the optimal solution, while for infeasible cases (✗), the accompanying number gives the total solving time in seconds. If the solver hit the time limit before finding an optimal solution, the cell is marked with *lim* and the best feasible solution is returned. Note that in the worst-case scenario the solver might fail to find a feasible solution or prove its infeasibility within the time limit.

Notably, our results show that in all considered cases the solver either found a falsifying trace or proved that no such trace exists within the imposed time limit, confirming that the framework is successful in generating load tests for all the encoded scenarios. Furthermore, solving times to the first feasible solution are generally much shorter than to the optimal one, showing that falsifying traces for many scenarios can be obtained within short time intervals.

We can further notice that the *free load shape* generally requires more time to be solved to optimality compared to the other load shapes. This is caused by the fact that the shape constraints imposed in the other template scenarios are effective in decreasing problem complexity.

In contrast to previous works relying on manual tuning (Straesser et al., 2023), the proposed technique automatically tunes parameters while guaranteeing their feasibility, eliminating the risk of configuration errors and the need for multiple test iterations. Moreover, the framework handles load shapes with multiple degrees of freedom that would be complex to configure manually, simplifying the practitioner's work and providing an advantage over earlier automatic test generation methods such as Barna et al. (2011), that are limited to less expressive

profiles. Finally, unlike search-based techniques (McMinn, 2011; Shen et al., 2015), our approach operates offline; once a proxy performance model is available, it requires no further costly interactions with the system.

Answer to Q2: The proposed method proves effective in automatically generating a diverse set of test cases within short time and with minimal manual configuration, or in demonstrating that none exist in the model, across a wide range of autoscaling policies, load shapes, and objectives.

6.5. *Q3: Test generation approaches comparison*

In this experiment, we address **Q3** by performing a two-fold evaluation. Firstly, we discuss approaches for generating workload traces by interacting with the system under test during execution, and discuss their applicability in practical scenarios. This evaluation will be conducted within the proposed modeling framework, since repeated online experiments on the system under test would be prohibitively costly. Secondly, we compare two model-based state exploration approaches, namely the proposed MILP-based solution procedure and a genetic algorithm (GA).

Experimental Setup. Here we consider the generation of tests for the application model described in §6.3.1 under the **VP** autoscaler implementation Eq. (8) (with $u^* = 0.5$). We generated workload traces of $T = 60$ time steps in order to violate the **under-provisioning** specification with $\tau = 1$. Moreover, we consider generating tests under the **free load shape**, mathematically encoded in Eq. (18).

We perform model-based load test generation through the following three approaches until a time limit of $\bar{t} = 300$ seconds for a total of 10 repetitions each:

Random Search. We randomly sample from the admissible workload set defined by Eq. (18) a workload trace $\mathbf{l}$ through hit-and-run sampling and compute the system evolution function $\mathbf{Y}(\mathbf{c}(0), \mathbf{l})$, that is then used to compute the value of the robustness function Eq. (15). The process is repeated until the time limit is hit. Whenever an improvement on the violation metric is found, we record the elapsed wall-clock time, the robustness value, and the number of evaluated samples.

Genetic Algorithm. We employ a genetic algorithm to explore the admissible workload set. We generate an initial population of 50 samples through hit-and-run sampling. In each generation, candidate workloads are evaluated using the same approach of the random search. After each mutation, the population is repaired by computing a projection of the workload trace on the admissible workload set. At the end of each generation, if an improvement on the violation metric is detected, we record the elapsed wall-clock time, the robustness value, and the current generation number. The process is repeated until the time limit is reached. Given that we consider a population size of 50, each generation involves the evaluation of a corresponding amount of samples.

MILP Encoding. We formulate the falsification problem as the MILP encoding (11) and solve it within the fixed time limit. In order to produce comparable results with the other methods, we set the parameters $\varepsilon = 0, \alpha = 1, \beta = 0$, thus the objective function corresponds to the violation metric. During the solving process, every time a feasible solution with a better objective value is found, we record the elapsed wall-clock time and the corresponding objective value.

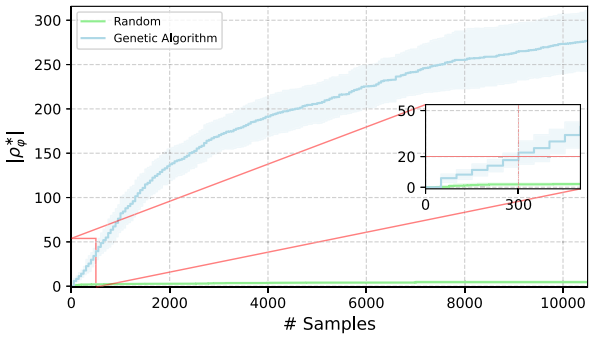

Fig. 8. Best violation versus number of evaluated samples for random search and GA. Solid lines show average values over multiple runs, with shaded areas representing standard deviations. The inset marks the tolerance value from previous experiments and the number of samples needed to reach it.

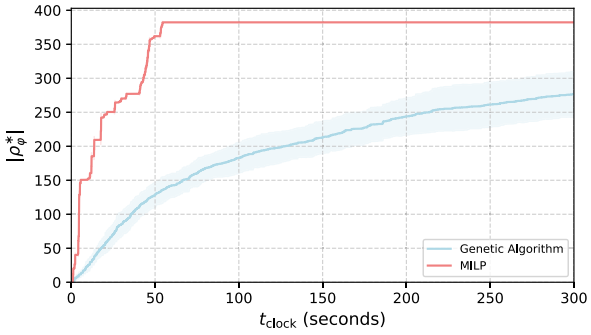

Fig. 9. Best violation versus wall-clock time for GA and MILP solver. Solid lines show average values over multiple runs, with shaded areas representing standard deviations.

Results. Fig. 8 reports the best violation magnitudes $|\rho^*_\varphi|$ obtained by random search and the genetic algorithm with respect to the number of evaluated samples. Notably, the naive random approach fails to find a satisfactory falsifying trace even after 10000 evaluations, thus resulting ineffective as an approach for load test generation in this scenario, whether applied to the model or directly on the system under test. The genetic algorithm is instead capable of refining the initially random samples and producing falsifying traces. As a reference, in §6.3.1 we generated traces considering a robustness tolerance value $\varepsilon = 20$. Since the best robustness values are updated every 50 samples in the genetic algorithm (corresponding to the population size), the inset shows that achieving a value of at least 20 to produce a falsifying trace according to tolerance constraints, requires an average of 300 samples.

Evaluating a single trace on the real system under test requires at best several minutes. Moreover, such executions must be repeated many times to identify a falsifying workload trace for a single generation scenario (i.e., a fixed set of load generation hyperparameters). These results confirm that sampling-based approaches over the system under test are impractical for systematic load test generation.

Since the genetic algorithm approach proved successful in generating falsifying traces within the proposed model, we compare it against the proposed MILP encoding in terms of wall-clock performance. Fig. 9 reports the best violation magnitudes $|\rho^*_\varphi|$ obtained by both approaches with respect to wall-clock time. The MILP solver did not reach a certificate of optimality within the time limit in any repetition, but consistently produced high quality solutions faster than the genetic algorithm, and with higher performance stability within different runs.

We statistically validated the comparative performance of the MILP and GA approaches across the 10 independent repetitions using one-sided Mann-Whitney U tests[4] (Mann and Whitney, 1947).

First, we compared the *wall-clock times to first violation* $t^v = \min\{t_{\text{clock}} \mid |\rho^*_\varphi(t_{\text{clock}})| \geq \varepsilon\}$ of the two approaches. We tested the null hypothesis $H_0 : \tilde{t}^v_{\text{MILP}} \geq \tilde{t}^v_{\text{GA}}$ against the alternative one $H_1 : \tilde{t}^v_{\text{MILP}} < \tilde{t}^v_{\text{GA}}$ with $\alpha = 0.01$. The test yielded a p-value $\approx 9.13 \times 10^{-5} < \alpha$ supporting the claim that MILP achieves the first violation faster than GA.

Secondly, we compared the *best violation magnitudes at time limit* $v^* = |\rho^*_\varphi(\bar{t})|$ of the two approaches. We tested the null hypothesis $H_0 : \tilde{v}^*_{\text{MILP}} \leq \tilde{v}^*_{\text{GA}}$ against the alternative one $H_1 : \tilde{v}^*_{\text{MILP}} > \tilde{v}^*_{\text{GA}}$ with $\alpha = 0.01$. The test yielded a p-value $\approx 3.19 \times 10^{-5} < \alpha$ confirming that the violation at time limit produced by the MILP approach is higher than that produced by GA.

Furthermore, we note that MILP solvers can also provide certificates of optimality or infeasibility, preventing wasted computational resources on search-based methods that may explore infeasible scenarios indefinitely without providing a non-existence verdict. These results support the choice of MILP-based techniques for solving the falsification problem in the proposed modeling framework.

Answer to Q3: Direct online falsification is shown to be impractical due to the excessive number of samples needed to identify a falsifying trace, multiplied by the computational cost of the evaluation of a single trace. Among model-based optimization techniques, the MILP encoding proved superior to genetic algorithm approaches.

6.6. Threats to validity

We discuss threats to the validity of the experimental evaluation, distinguishing between internal and external factors.

Internal Validity. Model mismatch may introduce discrepancies between queuing network predictions and the measured system behavior. We mitigate this risk through a sensitivity analysis experiment in §6.3.2 and by observing that measured violations consistently exceed model predictions in §6.3.3, which makes the test generation conservative. To reduce the impact of performance variability in cloud environments, each experiment is repeated multiple times and analyzed using non-parametric statistical tests.

External Validity. Our evaluation is conducted on controlled benchmarks rather than production deployments. Although the case study and autoscaler models reflect realistic designs and are widely used in related work (Incerto et al., 2023, 2025), they remain abstractions, and the results may not fully generalize to operational systems. Validation on real deployments is therefore left to future work. The simulation study also excludes over-provisioning scenarios. Since model mismatch typically occurs under saturation, and the specification tolerance ε filters minor violations, we do not expect false positives in this regime. Nonetheless, this scenario warrants further investigation.

7. Limitations

This section summarizes the main methodological limitations of the proposed approach and clarifies its scope of applicability.

[4] We used non-parametric Mann-Whitney U tests given the low sample number.

7.1. Modeling assumptions

In this paper, we focused on closed queuing network performance models for microservice applications to predict services throughput, as well as on rule-based and HPA-inspired autoscaler implementations. While the performance model choice is well established in the performance modeling literature, it cannot naturally capture all phenomena affecting performance in microservice applications. However, we argue that the model is not intended to provide definite performance prediction, but to support the generation of workload traces, that are ultimately validated on the system under test. Moreover, from a methodological standpoint, as discussed in Remarks 1 and 2, the proposed approach can in principle accommodate a wide variety of performance models and autoscaler policies, provided that PWA approximations of these components exist. Future work may investigate these modeling extensions, thus enabling the application of the approach to more complex systems and test scenarios. Furthermore, data-driven techniques (e.g., reinforcement learning Yamagata et al., 2021 or surrogate model learning (Bak et al., 2024)) could be employed when black-box models of the systems are available.

7.2. Optimization scalability

The MILP representation of the falsification problem (10) faces the complexity of NP-hard formulations, with worst-case exponential time complexity growth proportional to the number of microservices, time steps, and autoscaler PWA regions. However, regarding the test length, we argue that the main goal of the proposed methodology is the generation of short, focused test cases rather than the execution of long-duration sequences used in orthogonal approaches for expected usage validation (Calzarossa et al., 2016; Vögele et al., 2018). Regarding the system model complexity, we remark that solving times are not strictly predictable based on the number of binary variables, as computational effort depends heavily on the specific problem structure and the efficacy of solver heuristics. However, although the complexity of optimization problem (11) grows exponentially with the number of binary variables, modern MILP solvers are able to solve large problem instances thanks to decades of algorithmic and heuristic advances (Achterberg and Wunderling, 2013). Furthermore, despite this exponential growth, the problem is solved offline and does not impose real-time constraints; consequently, even computation times on the order of hours are negligible when compared to the expense of conducting large-scale live experiments. In our experiments in §6.5, instances up to $n = 10$ services and a horizon $T = 60$ time steps were consistently solved within a 5-minute timeout. Nevertheless, this limitation opens several avenues for future research, including the investigation of alternative solvers, such as SMT-based ones (e.g., Z3 de Moura and Bjørner, 2008, dReal (Gao et al., 2013)), beyond the heuristic method evaluated in §6.5.

7.3. Modeling effort

The proposed methodology depends on the availability of a sufficiently accurate performance model. While building a reliable performance model requires significant upfront effort, we argue that this investment offers a favorable trade-off compared to purely online testing approaches. First, in §6.5 we show that finding falsifying traces online through online state-space exploration is infeasible in high-dimensional load configuration spaces. Furthermore, after the initial model inference, the model can be used to generate test cases across different configurations, as demonstrated in §6.4, while an online approach might need to start from scratch. Finally, this modeling effort also unlocks additional model-based microservice engineering techniques, such as autoscaling (Incerto et al., 2018), what-if analysis (Kowal et al., 2015), and capacity planning (Incerto et al., 2021).

Furthermore, model mismatch may affect the estimation of the specification violations. In §5.4, we discuss mitigation strategies to reduce

the risk of false positives, including the adoption of robust specification violation thresholds (see tolerance parameters ϵ in §5.1). Additionally, whenever a modeling error affects the applicability of the methodology, generated test cases can be used to refine the model. Nevertheless, empirical results confirm that, despite prediction errors, the methodology successfully exposes performance failures during execution. Notably, in the under-provisioning specification, actual performance violations were more severe than those predicted. This indicates that, although QN models cannot capture all dynamics of the real system, they still are adequate as proxy models for effective load test generation within the proposed approach.

8. Conclusion

This paper presented a novel approach for load test generation in elastic microservice applications. The contribution of the paper lies in a structured, model-based offline approach that complements classical expected usage load tests. An end-to-end workflow to evaluate the outcome of the test generation procedure is also presented.

The effectiveness of our approach has been validated on an example elastic microservice application whose performance was modeled as a closed QN. We showed that the test generation process unveils performance failures despite model discrepancies in the system equipped with an autoscaler. Moreover, we demonstrated that the method is effective in generating a diverse set of test cases under different operating conditions within a short amount of time and with minimal manual configuration.

Future work could address the computational scalability of the test generation approach by exploring alternative solution methods, enabling its usage in more complex scenarios. From a software engineering perspective, building upon the evaluation pipeline proposed in §5.4, another direction for future work is the development of a production-ready implementation of the proposed method into CI/CD pipelines.

CRediT authorship contribution statement

Marco Zamponi: Writing – review & editing, Writing – original draft, Visualization, Software, Methodology, Investigation, Conceptualization; **Daniele Masti:** Writing – review & editing, Writing – original draft, Validation, Supervision, Funding acquisition, Formal analysis, Conceptualization; **Emilio Incerto:** Writing – review & editing, Validation, Supervision, Investigation, Funding acquisition, Conceptualization; **Franco Raimondi:** Writing – review & editing, Validation, Supervision, Investigation, Funding acquisition, Conceptualization; **Mirco Tribastone:** Writing – review & editing, Validation, Supervision, Investigation, Funding acquisition, Conceptualization.

Data availability

The code of the tool and the scripts used for our experiments are available at https://doi.org/10.5281/zenodo.17938632

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Daniele Masti reports financial support was provided by Ministero dell'Universitá e della Ricerca. Emilio Incerto reports financial support was provided by Ministero dell'Universitá e della Ricerca. Mirco Tribastone reports financial support was provided by Ministero dell'Universitá e della Ricerca. Franco Raimondi reports financial support was provided by European Commission. If there are other authors, they 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 has been partially funded by the European Union - NextGenerationEU under the Italian Ministry of University and Research (MUR) through the National Innovation Ecosystem grant ECS00000041 - VITALITY (CUP: D13C21000430001), the SERICS project (SEcurity and RIghts in the CyberSpace, PE0000014, PNRR M4C2 I.1.3), and the 'Resilienza Economica e Digitale' project (CUP: D67G23000060001), awarded as part of the 'Department of Excellence' initiative (Dipartimenti di Eccellenza 2023–2027, Ministerial Decree no. 230/2022). It has also received partial funding from the European HORIZON-KDT-JU research project MATISSE *Model-based engineering of Digital Twins for early verification and validation of Industrial Systems*, under the HORIZON-KDT-JU-2023-2-RIA program (Proposal number: 101140216-2, KDT232RIA_00017). Additional support was provided by project RDS - PTR22-24 P2.1 Cybersecurity. Daniele Masti is also part of INdAM/GNAMPA.

Appendix A. Load Shapes

In this section we describe the constraints defining the admissible workload set $\mathcal{L}$ of each load shape presented in §5.3.

Ramp Load Shape.

$$\begin{aligned} \underline{l} \le l(t) \le \bar{l} \quad & \forall t = 1{:}T \\ l(t) = l(t-1) \quad & \forall t = 2{:}\tau_1 \\ l(t) = l(t-1) + \alpha \quad & \forall t = \tau_1{+}1{:}\tau_2 \\ l(t) = l(t-1) \quad & \forall t = \tau_2{+}1{:}T \\ |\alpha| \le \sigma, & \end{aligned}$$

with $\tau_1 = \lfloor 0.25T \rfloor$, $\tau_2 = \lfloor 0.75T \rfloor$ denoting respectively the start and the end of the ramp phase. The parameter α defines the constant slope of the ramp which absolute value is bounded by the scalar parameter σ.

Spike Load Shape.

$$\begin{aligned} \underline{l} \le l(t) \le \bar{l} \quad & \forall t = 1{:}T \\ l(t) = l(t-1) \quad & \forall t = 2{:}\tau_1 \\ l(t) = l(t-1) + \alpha \quad & \forall t = \tau_1{+}1{:}\tau_2 \\ l(t) = \gamma\, l(t-1) \quad & \forall t = \tau_2{+}1{:}\tau_3 \\ l(t) = l(t-1) \quad & \forall t = \tau_3{+}1{:}T \\ \alpha \in \mathbb{R}, \quad 0 \le \gamma \le 0.9 & \end{aligned}$$

with $\tau_1 = \lfloor 0.15T \rfloor$, $\tau_2 = \lfloor 0.25T \rfloor$, and $\tau_3 = \lfloor 0.5T \rfloor$, denoting respectively the end of the warm-up period, the end of the peak growth phase, and the end of the exponential decay. The parameter α controls the magnitude of the spike growth, while γ regulates the exponential decay rate.

Sawtooth Load Shape.

$$\begin{aligned} \underline{l} \le l(t) \le \bar{l} \quad & \forall t = 1{:}T \\ l(t) = l(t-1) \quad & \forall t = 2{:}\left\lfloor \tfrac{C}{2} \right\rfloor \\ 1 \le l(t) - l(t-1) \le \sigma \quad & \forall t = b_i{+}1{:}b_i{+}R, \\ & \forall i = 0{:}N{-}1 \\ -\sigma \le l(t) - l(t-1) \le -1 \quad & \forall t = b_i{+}R{+}1{:}b_i{+}C, \\ & \forall i = 0{:}N{-}1 \\ l(t) = l(t-1) \quad & \forall t = \left\lfloor \tfrac{C}{2} \right\rfloor{+}NC{+}1{:}T, \end{aligned}$$

with $N = 2$ denoting the number of sawtooth cycles, $r = 0.6$ the ratio of ramp-up length to cycle length, $C = \lfloor T/(N+1) \rfloor$ the cycle duration, $R = \lfloor rC \rfloor$ the ramp-up duration, and $b_i = \lfloor C/2 \rfloor + iC$ the start of each cycle. Each cycle is characterized by a bounded ramp-up followed by a bounded ramp-down, while the sequence remains constant outside of the cycle intervals. Positive scalar σ denotes the bound on the rate of change.

References

Achterberg, T., Wunderling, R., 2013. Mixed integer programming: analyzing 12 years of progress. In: Jünger, M., Reinelt, G. (Eds.), Facets of Combinatorial Optimization: Festschrift for Martin Grötschel. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 449–481. https://doi.org/10.1007/978-3-642-38189-8_18

Agos Jawaddi, S.N., Ismail, A., Mohammad Hatta, M. N.H., Kamarulzaman, A.F., 2024. Insights into cloud autoscaling: a unique perspective through MDP and DTMC formal models. J. Supercomput. 80 (4), 5073–5107. https://doi.org/10.1007/s11227-023-05665-7

Ágoston, K.C., Marianna, E., 2024. Mixed integer linear programming formulation for K-means clustering problem. Cent. Eur. J. Oper. Res. 32 (1), 11–27. https://doi.org/10.1007/s10100-023-00881-1

Ari, I., Hong, B., Miller, E.L., Brandt, S.A., Long, D. D.E., 2003. Managing flash crowds on the internet. In: 11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer Telecommunications Systems, 2003. MASCOTS 2003.. IEEE, pp. 246–249. https://doi.org/10.1109/MASCOT.2003.1240667

Amazon Web Services, Inc. Auto Scaling Documentation. https://docs.aws.amazon.com/autoscaling/.

Bak, S., Bogomolov, S., Hekal, A., Kochdumper, N., Lew, E., Mata, A., Rahmati, A., 2024. Falsification using reachability of surrogate koopman models. In: Proceedings of the 27th ACM International Conference on Hybrid Systems: Computation and Control. Association for Computing Machinery, New York, NY, USA, pp. 1–13. https://doi.org/10.1145/3641513.3650141

Baresi, L., Guinea, S., Leva, A., Quattrocchi, G., 2016. A discrete-time feedback controller for containerized cloud applications. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM, Seattle WA USA, pp. 217–228. https://doi.org/10.1145/2950290.2950328

Barna, C., Litoiu, M., Ghanbari, H., 2011. Autonomic load-testing framework. In: Proceedings of the 8th ACM International Conference on Autonomic Computing. ACM, Karlsruhe Germany, pp. 91–100. https://doi.org/10.1145/1998582.1998598

Bartocci, E., Deshmukh, J., Donzé, A., Fainekos, G., Maler, O., Ničković, D., Sankaranarayanan, S., 2018. Specification-based monitoring of cyber-physical systems: a survey on theory, tools and applications. In: Bartocci, E., Falcone, Y. (Eds.), Lectures on Runtime Verification: Introductory and Advanced Topics. Springer International Publishing, Cham, pp. 135–175. https://doi.org/10.1007/978-3-319-75632-5_5

Bayan, M., Cangussu, J.W., 2008. Automatic feedback, control-based, stress and load testing. In: Proceedings of the 2008 ACM Symposium on Applied Computing. Association for Computing Machinery, New York, NY, USA, pp. 661–666. https://doi.org/10.1145/1363686.1363847

Bemporad, A., 2023. A piecewise linear regression and classification algorithm with application to learning and model predictive control of hybrid systems. IEEE Trans. Automat. Contr. 68 (6), 3194–3209. https://doi.org/10.1109/TAC.2022.3183036

Bemporad, A., Morari, M., 1999. Control of systems integrating logic, dynamics, and constraints. Automatica 35 (3), 407–427. https://doi.org/10.1016/S0005-1098(98)00178-2

Bemporad, A., Morari, M., Dua, V., Pistikopoulos, E.N., 2000a. The explicit solution of model predictive control via multiparametric quadratic programming. In: Proceedings of the 2000 American Control Conference. ACC (IEEE Cat. No.00ch36334). Vol. 2, pp. 872–876 vol.2. https://doi.org/10.1109/ACC.2000.876624

Bemporad, A., Torrisi, F.D., Morari, M., 2000b. Optimization-based verification and stability characterization of piecewise affine and hybrid systems. In: Lynch, N., Krogh, B.H. (Eds.), Hybrid Systems: Computation and Control. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 45–58. https://doi.org/10.1007/3-540-46430-1_8

Bortolussi, L., Hillston, J., Latella, D., Massink, M., 2013. Continuous approximation of collective system behaviour: a tutorial. Perform. Eval. 70 (5), 317–349. https://doi.org/10.1016/j.peva.2013.01.001

Calzarossa, M.C., Massari, L., Tessera, D., 2016. Workload characterization: a survey revisited. ACM Comput. Surv. 48 (3), 1–43. https://doi.org/10.1145/2856127

Cooper, Q., Krishnamurthy, D., Amannejad, Y., 2024. Budget aware performance test selection for microservices. In: 2024 IEEE 17th International Conference on Cloud Computing (CLOUD), pp. 376–385. https://doi.org/10.1109/CLOUD62652.2024.00049

de Moura, L., Bjørner, N., 2008. Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (Eds.), Tools and Algorithms for the Construction and Analysis of Systems. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

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. https://doi.org/10.1007/978-3-319-67425-4_12

Dutreilh, X., Moreau, A., Malenfant, J., Rivierre, N., Truck, I., 2010. From data center resource allocation to control theory and back. In: 2010 IEEE 3rd International Conference on Cloud Computing. IEEE, pp. 410–417. https://doi.org/10.1109/CLOUD.2010.55

Evangelidis, A., Parker, D., Bahsoon, R., 2018. Performance modelling and verification of cloud-based auto-scaling policies. Future Gen. Comput. Syst. 87, 629–638. https://doi.org/10.1016/j.future.2017.12.047

Fabiani, F., Stellato, B., Masti, D., Goulart, P.J., 2025. A neural network-based approach to hybrid systems identification for control. Automatica 174, 112130. https://doi.org/10.1016/j.automatica.2025.112130

Fowler, M., 2012. Patterns of Enterprise Application Architecture. Addison-Wesley.

Gambi, A., Hummer, W., Dustdar, S., 2013. Testing elastic systems with surrogate models. In: 2013 1st International Workshop on Combining Modelling and Search-Based Software Engineering (CMSBSE), pp. 8–11. https://doi.org/10.1109/CMSBSE.2013.6604429

Gandhi, A., Dube, P., Karve, A., Kochut, A., Zhang, L., 2018. Model-driven optimal resource scaling in cloud. Softw. Syst. Model. 17 (2), 509–526. https://doi.org/10.1007/s10270-017-0584-y

Gao, S., Kong, S., Clarke, E.M., 2013. Dreal: an SMT solver for nonlinear theories over the reals. In: Bonacina, M.P. (Ed.), Automated Deduction – CADE-24. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 208–214. https://doi.org/10.1007/978-3-642-38574-2_14

García, C.E., Prett, D.M., Morari, M., 1989. Model predictive control: theory and practice—a survey. Automatica 25 (3), 335–348. https://doi.org/10.1016/0005-1098(89)90002-2

Ghanbari, H., Simmons, B., Litoiu, M., Barna, C., Iszlai, G., 2012. Optimal autoscaling in a iaas cloud. In: Proceedings of the 9th International Conference on Autonomic Computing. ACM, San Jose California USA, pp. 173–178. https://doi.org/10.1145/2371536.2371567

Giamattei, L., Guerriero, A., Malavolta, I., Mascia, C., Pietrantuono, R., Russo, S., 2024. Identifying performance issues in microservice architectures through causal reasoning. In: Proceedings of the 5th ACM/IEEE International Conference on Automation of Software Test (AST 2024). ACM, Lisbon Portugal, pp. 149–153. https://doi.org/10.1145/3644032.3644460

Golshani, E., Ashtiani, M., 2021. Proactive auto-scaling for cloud environments using temporal convolutional neural networks. J. Parallel Distrib. Comput. 154, 119–141. https://doi.org/10.1016/j.jpdc.2021.04.006

Gordon, W.J., Newell, G.F., 1967. Closed queuing systems with exponential servers. Oper. Res. 15 (2), 254–265. https://doi.org/10.1287/opre.15.2.254

Gunawi, H.S., Hao, M., Suminto, R.O., Laksono, A., Satria, A.D., Adityatama, J., Eliazar, K.J., 2016. Why does the cloud stop computing?: lessons from hundreds of service outages. In: Proceedings of the Seventh ACM Symposium on Cloud Computing. ACM, Santa Clara CA USA, pp. 1–16. https://doi.org/10.1145/2987550.2987583

Gurobi Optimization, L., 2024. Gurobi optimizer reference manual.

Heemels, W.P.M.H., De Schutter, B., Bemporad, A., 2001. Equivalence of hybrid dynamical models. Automatica 37 (7), 1085–1091. https://doi.org/10.1016/S0005-1098(01)00059-0

Herbst, N.R., Kounev, S., Reussner, R., 2013. Elasticity in cloud computing: what it is, and what it is not. In: 10th International Conference on Autonomic Computing (ICAC 13), pp. 23–27.

Herbst, N.R., Kounev, S., Weber, A., Groenda, H., 2015. BUNGEE: An elasticity benchmark for self-Adaptive iaas cloud environments. In: 2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, pp. 46–56. https://doi.org/10.1109/SEAMS.2015.23

Hinton, A., Kwiatkowska, M., Norman, G., Parker, D., 2006. PRISM: A tool for automatic verification of probabilistic systems. In: Hermanns, H., Palsberg, J. (Eds.), Tools and Algorithms for the Construction and Analysis of Systems. Springer Berlin Heidelberg, Berlin, Heidelberg. Vol. 3920, pp. 441–444. https://doi.org/10.1007/11691372_29

IEEE, 1990. Standard glossary of software engineering terminology. IEEE Std 610.12-1990 1–84.

Incerto, E., Napolitano, A., Tribastone, M., 2021. Learning queuing networks via linear optimization. In: Proceedings of the ACM/SPEC International Conference on Performance Engineering. ACM, Virtual Event France, pp. 51–60. https://doi.org/10.1145/3427921.3450245

Incerto, E., Pizziol, R., Russo, G.R., Tribastone, M., 2025. Wasteless: an optimal provisioner for self-adaptive second-generation serverless applications. In: 2025 IEEE/ACM 20th Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS), pp. 61–72. https://doi.org/10.1109/SEAMS66627.2025.00015

Incerto, E., Pizziol, R., Tribastone, M., 2023. μOpt: an efficient optimal autoscaler for microservice applications. In: 2023 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS), pp. 67–76. https://doi.org/10.1109/ACSOS58161.2023.00024

Incerto, E., Tribastone, M., Trubiani, C., 2017. Software performance self-adaptation through efficient model predictive control. In: 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 485–496. https://doi.org/10.1109/ASE.2017.8115660

Incerto, E., Tribastone, M., Trubiani, C., 2018. Combined vertical and horizontal autoscaling through model predictive control. In: Aldinucci, M., Padovani, L., Torquati, M. (Eds.), Euro-Par 2018: Parallel Processing. Springer International Publishing, Cham. Vol. 11014, pp. 147–159. https://doi.org/10.1007/978-3-319-96983-1_11

Iqbal, W., Erradi, A., Mahmood, A., 2018. Dynamic workload patterns prediction for proactive auto-scaling of web applications. J. Network Comput. Appl. 124, 94–107. https://doi.org/10.1016/j.jnca.2018.09.023

Ireland, D., 2020. Milliseconds Make Millions. https://www.deloitte.com/ie/en/services/consulting/research/milliseconds-make-millions.html.

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

Kowal, M., Tschaikowski, M., Tribastone, M., Schaefer, I., 2015. Scaling size and parameter spaces in variability-Aware software performance models (t). In: 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 407–417. https://doi.org/10.1109/ASE.2015.16

Kubernetes, Horizontal Pod Autoscaling, 2026. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/.

Lindemann, L., Dimarogonas, D.V., 2019. Robust control for signal temporal logic specifications using discrete average space robustness. Automatica 101, 377–387. https://doi.org/10.1016/j.automatica.2018.12.022

Lorido-Botran, T., Miguel-Alonso, J., Lozano, J.A., 2014. A review of auto-scaling techniques for elastic applications in cloud environments. J. Grid Comput. 12 (4), 559–592. https://doi.org/10.1007/s10723-014-9314-7

Malone, D., Duffy, K., Leith, D., 2007. Modeling the 802.11 distributed coordination function in nonsaturated heterogeneous conditions. IEEE/ACM Trans. Networking 15 (1), 159–172. https://doi.org/10.1109/TNET.2006.890136

Mann, H.B., Whitney, D.R., 1947. On a test of whether one of two random variables is stochastically larger than the other. Annals Math. Stat. 18 (1), 50–60. 2236101.

Mascia, C., Guerriero, A., Giamattei, L., Pietrantuono, R., Russo, S., 2025. Microservices performance testing with causality-enhanced large language models. In: 2025 IEEE/ACM Second International Conference on AI Foundation Models and Software Engineering (Forge), pp. 136--140. https://doi.org/10.1109/Forge66646.2025.00022

Mathesen, L., Pedrielli, G., Fainekos, G., 2021. Efficient optimization-based falsification of cyber-physical systems with multiple conjunctive requirements. In: 2021 IEEE 17th International Conference on Automation Science and Engineering (CASE). IEEE, pp. 732–737. https://doi.org/10.1109/CASE49439.2021.9551474

McMinn, P., 2011. Search-based software testing: past, present and future. In: 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops, pp. 153–163. https://doi.org/10.1109/ICSTW.2011.100

Megahed, A., Mohamed, M., Tata, S., 2017. A stochastic optimization approach for cloud elasticity. In: 2017 IEEE 10th International Conference on Cloud Computing (CLOUD), pp. 456–463. https://doi.org/10.1109/CLOUD.2017.65

Pabla, C.S., 2009. Completely fair scheduler. Linux J. 2009 (184), 4.

Papadopoulos, A.V., Ali-Eldin, A., Årzén, K.-E., Tordsson, J., Elmroth, E., 2016. PEAS: A performance evaluation framework for auto-Scaling strategies in cloud applications. ACM Trans. Model. Perform. Eval. Comput. Syst. 1 (4), 15:1–15:31. https://doi.org/10.1145/2930659

Quattrocchi, G., Incerto, E., Pinciroli, R., Trubiani, C., Baresi, L., 2024. Autoscaling solutions for cloud applications under dynamic workloads. IEEE Trans. Serv. Comput. 17 (3), 804–820. https://doi.org/10.1109/TSC.2024.3354062

Serracanta, B., Lukács, A., Rodriguez-Natal, A., Cabellos, A., Rétvári, G., 2025. On the stability of the kubernetes horizontal autoscaler control loop. IEEE Access 13, 7160–7166. https://doi.org/10.1109/ACCESS.2025.3526751

Shen, D., Luo, Q., Poshyvanyk, D., Grechanik, M., 2015. Automating performance bottleneck detection using search-based application profiling. In: Proceedings of the 2015 International Symposium on Software Testing and Analysis. ACM, Baltimore MD USA, pp. 270–281. https://doi.org/10.1145/2771783.2771816

Spinner, S., Casale, G., Brosig, F., Kounev, S., 2015. Evaluating approaches to resource demand estimation. Perform. Eval. 92, 51–71. https://doi.org/10.1016/j.peva.2015.07.005

Straesser, M., Eismann, S., von Kistowski, J., Bauer, A., Kounev, S., 2023. Autoscaler evaluation and configuration: a practitioner's guideline. In: Proceedings of the 2023 ACM/SPEC International Conference on Performance Engineering. Association for Computing Machinery, New York, NY, USA, pp. 31–41. https://doi.org/10.1145/3578244.3583721

Straesser, M., Grohmann, J., von Kistowski, J., Eismann, S., Bauer, A., Kounev, S., 2022. Why is it not solved yet? challenges for production-Ready autoscaling. In: Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering. Association for Computing Machinery, New York, NY, USA, pp. 105–115. https://doi.org/10.1145/3489525.3511680

Tjeng, V., Xiao, K., Tedrake, R., 2017. Evaluating robustness of neural networks with mixed integer programming. https://doi.org/10.48550/arXiv.1711.07356

Tong, J., Wei, M., Pan, M., Yu, Y., 2021. A holistic auto-Scaling algorithm for multi-Service applications based on balanced queuing network. In: 2021 IEEE International Conference on Web Services (ICWS), pp. 531–540. https://doi.org/10.1109/ICWS53863.2021.00074

Velepucha, V., Flores, P., 2023. A survey on microservices architecture: principles, patterns and migration challenges. IEEE Access 11, 88339–88358. https://doi.org/10.1109/ACCESS.2023.3305687

Vilaplana, J., Solsona, F., Teixidó, I., Mateo, J., Abella, F., Rius, J., 2014. A queuing theory model for cloud computing. J. Supercomput. 69 (1), 492–507. https://doi.org/10.1007/s11227-014-1177-y

Vitui, A., Chen, T.-H., 2024. MLOLET - Machine learning optimized load and endurance testing: an industrial experience report. In: Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering. Association for Computing Machinery, New York, NY, USA, pp. 1956–1966. https://doi.org/10.1145/3691620.3695258

Vögele, C., van 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. https://doi.org/10.1007/s10270-016-0566-5

Wilcoxon, F., 1945. Individual comparisons by ranking methods. Biometr. Bull. 1 (6), 80–83. 3001968.

Yamagata, Y., Liu, S., Akazaki, T., Duan, Y., Hao, J., 2021. Falsification of cyber-Physical systems using deep reinforcement learning. IEEE Trans. Software Eng. 47 (12), 2823–2840. https://doi.org/10.1109/TSE.2020.2969178

Zou, D., Lu, W., Zhu, Z., Lu, X., Zhou, J., Wang, X., Liu, K., Wang, K., Sun, R., Wang, H., 2024. Optscaler: a collaborative framework for robust autoscaling in the cloud. Proc. VLDB Endowment 17 (12), 4090–4103. https://doi.org/10.14778/3685800.3685829