HQP: Hybrid Queueing and Configuration-Aware Performance Modeling for Microservice Systems

YUANJIE XIA, University of Waterloo, Canada
KUNDI YAO, University of Waterloo, Canada
LIZHI LIAO, University of Guelph, Canada
DANIELE DI POMPEO, University of L'Aquila, Italy
CATIA TRUBIANI, Gran Sasso Science Institute, Italy
BORIS ZIBITSKER, BEZNext, USA
WEIYI SHANG, University of Waterloo, Canada

Understanding the performance of microservice-based systems is challenging because workloads and configurations often change dynamically, and their combined effects can influence system behavior. Existing performance testing and modeling techniques study workloads or configurations in isolation, overlooking their joint impact. The large configuration space and diverse workload patterns further complicate accurate prediction. Although prior studies have combined performance models with queueing formalisms, limited work has integrated configuration-aware modeling with queueing analysis to capture workload-dependent, system-level dynamics. We present HQP (**H**ybrid **Q**ueueing and Configuration-aware **P**erformance Modeling), a modeling approach that integrates lightweight configuration-aware performance models with the Queueing Petri Net (QPN) formalism to capture both service-level behavior and architectural interactions. HQP models how configuration settings and workload variations interact and propagate through microservice dependencies. Using the μBench, which allows customizable architectures and service functions, we construct four microservice systems to evaluate HQP. Results show that HQP achieves Spearman correlations up to 0.94 and MARE below 11%, while requiring only one-fifth the samples needed by baselines. We further introduce SCPM (**S**earch and **C**ondition **P**rediction on hybrid **M**odeling), a search-based method that identifies performance-equivalent, performance-spiked, and performance-saturated conditions with over 80% accuracy. Together, HQP and SCPM provide efficient, accurate performance analysis for evolving microservice systems.

CCS Concepts: • **Software and its engineering → Software performance**.

Additional Key Words and Phrases: Performance Modeling, Software Architecture, Performance Testing, Microservices

ACM Reference Format:
Yuanjie Xia, Kundi Yao, Lizhi Liao, Daniele Di Pompeo, Catia Trubiani, Boris Zibitsker, and Weiyi Shang. 2025. HQP: Hybrid Queueing and Configuration-Aware Performance Modeling for Microservice Systems. *ACM Trans. Softw. Eng. Methodol.* 1, 1 (August 2025), 36 pages. https://doi.org/XXXXXXX.XXXXXXX

Authors' Contact Information: Yuanjie Xia, yuanjie.xia@uwaterloo.ca, University of Waterloo, Waterloo, Ontario, Canada; Kundi Yao, University of Waterloo, Waterloo, Canada, kundi.yao@uwaterloo.ca; Lizhi Liao, University of Guelph, St. John's, Canada, lizhi.liao@uoguelph.ca; Daniele Di Pompeo, University of L'Aquila, L'Aquila, Italy, daniele.dipompeo@univaq.it; Catia Trubiani, Gran Sasso Science Institute, L'Aquila, Italy, catia.trubiani@gssi.it; Boris Zibitsker, BEZNext, Chicago, USA, bzibitsker@beznext.com; Weiyi Shang, University of Waterloo, Waterloo, Canada, wshang@uwaterloo.ca.

ACM 1557-7392/2025/8-ART
https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Microservices have become a popular architectural style for building modern applications [42], since developers can focus on small, independent services that can be developed, deployed, and scaled separately. This design improves modularity, making it easier to maintain and extend systems [48]. Many real systems, such as Netflix or Uber, use hundreds of microservices that run across many instances[1]. While this architecture offers flexibility, it also creates new research challenges, e.g., in performance evaluation. Small changes in one service (i.e., microservice) can affect many others because services are closely connected through APIs and shared resources. As a result, it is difficult to keep the system stable and fast as it evolves.

Performance assurance is one of the most critical and challenging aspects of microservice systems. Developers can easily test each service independently, but ensuring end-to-end performance across all services is much more complex. A minor code or configuration change in one service may have a negligible local effect but can cause a substantial impact on the system's end-to-end performance [3]. This complexity arises because the performance of one service depends on how fast others respond and how the overall workload flows through the system. Since each service can be updated or tuned by different teams, predicting the cascading effects of such changes is extremely difficult before deployment. Furthermore, in production environments, systems face continuously varying user demands and operational conditions, making static testing approaches insufficient for capturing real-world performance behavior.

To understand these effects without running every possible test, researchers build **performance models** that can predict the system's behavior under different conditions [63]. These models abstract system behavior and enable estimation of performance metrics (e.g., latency or throughput) under various circumstances. To the best of our knowledge, prior work on performance modeling has primarily focused on two key factors (i.e., workload and configuration changes) that influence system performance, but has considered them in isolation.

Workload changes represent what users send to the system, including the number of requests, request types, data sizes, and traffic patterns. Previous studies have developed workload-aware performance models to address this factor. Fuzzing-based testing [31] generates diverse workload inputs to detect performance regressions and edge cases. Workload replay methods [65] reproduce traces from production environments to study real performance behavior under authentic usage patterns. Queueing network performance models have been adopted in [47] to quantify the impact of heterogeneous workloads on the performance characteristics of microservices' design patterns. Other approaches [35, 36, 38, 52] compare workloads across different system versions to identify performance degradations.

Configuration changes define how the system is set up, including parameters such as CPU limits, memory allocations, thread pool sizes, caching settings, and timeout values. Configuration-aware models study how these system parameters influence performance characteristics [20, 46, 55]. These models are often constructed using machine learning techniques or analytical methods and can predict how modifications to specific configuration options affect throughput, latency, or resource utilization. Such models help developers understand the performance implications of tuning decisions and identify optimal configurations for specific objectives.

Although both workload-aware and configuration-aware performance modeling have been extensively studied, existing methods face fundamental limitations: (1) *Single-factor focus*: they typically model only one factor at a time (either workload or configuration changes) while holding the other constant. For instance, a configuration that performs well under a certain load may become a bottleneck in the case of concurrent and heterogeneous loads, or a workload pattern

[1]Microservices examples: https://blog.dreamfactory.com/microservices-examples

that runs efficiently with default settings may degrade significantly when parameters are modified; (2) *Missing architectural awareness*: current models often treat the system as a black box, without decomposing performance into per-service contributions or capturing how workloads propagate through service dependencies. This omission limits reusability across deployments and obscures which services may represent the root cause for performance degradation. Architecture-based modeling tools using Queueing Petri Nets (QPNs) [29, 34, 37] or Layered Queueing Networks (LQNs) [12, 53] can describe service structures and dependencies, but they primarily focus on either workload or code changes rather than dynamic configuration behavior; (3) *Combinatorial explosion and high modeling cost*: testing all combinations of workloads and configurations is impractical due to the exponential growth of the test space, and existing sampling methods do not strategically explore the configuration-workload space to identify risky performance conditions (e.g., saturation points, performance spikes, or performance-equivalent scenarios). Furthermore, building accurate joint models typically requires extensive sampling across both dimensions, leading to prohibitively high testing and training costs for large-scale systems with numerous services and configuration options. **As a result, there is no practical model that can jointly account for configuration and workload variations while capturing architectural dependencies in large microservice systems.** Although queueing networks can describe service structures and dependencies, no prior research has integrated these performance models for the system's end-to-end performance prediction. The risk is that performance predictions at the system level can be significantly inaccurate when either factor changes, leading to unexpected slowdowns, bottlenecks, or resource wastage that are difficult to foresee and diagnose.

To address this problem, we propose HQP (**H**ybrid **Q**ueueing and Configuration-aware **P**erformance Modeling), an approach that integrates lightweight configuration-aware performance models with a queueing model. Our goal is to accurately and efficiently estimate system performance as both configurations and workloads change simultaneously. The approach works in two stages:

- **Building models for individual services.** We first build performance models for each microservice type locally using the CoMSA sampling method [64]. CoMSA efficiently selects configuration samples and trains a model that captures how each service's performance (e.g., latency or throughput) changes under different settings and workloads. This addresses challenge (3) by modeling at the service level with reduced sampling costs.
- **Building a hybrid model based on the system architecture model.** We then combine these service-level models with an architecture model using the Queueing Petri Net (QPN) [29]. The QPN model captures how requests flow between services and how queues, synchronization, and communication delays affect system behavior. This hybrid model allows us to estimate end-to-end performance for any combination of workloads and configurations. This addresses challenges (1), (2), and (3) by enabling system-wide composition and modeling workload propagation through service dependencies.

To evaluate our approach, we use μBench [8], an application that enables users to customize both the system architecture and the functions of its microservices. Therefore, we adopt a simple (TrainTicket-style) architecture and a complex (TeaStore-style) architecture as the reference architecture of our systems. The architectural complexity discussed here primarily refers to the system's topology, as the system's overall functionality is derived from the workload under analysis, which is part of the performance evaluation. For each architecture, we further explore different real-world scenarios by creating two variants: (1) customized workload type, μBench allows users to modify service invocations and the configuration settings in each service under different workloads; (2) unified and diverse setups on service types within the microservices. The unified setup is a system in which all services are implemented using a single tool, whereas the diverse setup is one in which

services are implemented using a mix of tools. The platform lets us precisely control workloads and configurations.

As discussed above, our approach is built on accurate performance models for each microservice type. If the performance of a tool cannot be precisely predicted in a local environment, it is impossible to build an accurate performance model of the entire system. As a result, we select two tools, namely FFmpeg and lrzip, that have been shown to build accurate performance models in prior work [20, 26, 43, 64]. Four widely used configuration-aware methods, such as XGBoost [6], DeepPerf [20], DaL [15], and HyPerf [10], serve as strong baselines for building highly accurate performance models for our selected tools.

We evaluate our approach based on two key criteria: accuracy and cost. Our results show that the proposed method achieves high prediction accuracy, with Spearman correlations up to 0.94 and mean absolute relative errors (MARE) below 11%. At the same time, it reduces the total modeling and computation cost by up to 90% compared to baseline methods.

With our effective and efficient model, we further explore a proactive search-based method for performance conditions that are useful and important in real scenarios. We develop a tool called SCPM (**S**earch and **C**ondition **P**rediction on hybrid **M**odeling). It automatically mutates configuration and workload parameters to find cases that trigger unusual performance behaviors. This addresses challenges (1) and (3) by strategically exploring the joint configuration-workload space to detect three key performance conditions that prior work seeks to identify:

- performance-equivalent, where performance does not change noticeably.
- performance-spiked, where short performance spikes occur.
- performance-saturated, where the system reaches its limits or bottlenecks.

SCPM also detects hundreds of configuration-workload cases that match the three performance conditions, with around 80% classification accuracy. These results show that combining architecture-based modeling and fuzzing exploration enables efficient and accurate analysis of large configuration-workload spaces. The approach provides practical benefits for performance assurance in real microservice systems, where testing every configuration or workload combination is infeasible.

In summary, this paper makes the following contributions:

- We propose a hybrid modeling method, HQP, which offers a cost-efficient approach to predicting the performance of microservice systems. HQP integrates configuration-aware performance models with a queueing model to accurately estimate system behavior under combined changes in both configuration and workload.
- We introduce SCPM, a search-based exploration method that automatically identifies performance-equivalent, performance-spiked, and performance-saturated conditions.
- We quantitatively evaluate our method using real microservice architectures and demonstrate that it achieves higher accuracy and lower cost than existing configuration-aware modeling baselines.

Paper Organization. Section 2 reviews existing performance modeling techniques related to configurations, workloads, and queueing-based methods. Section 3 outlines the challenges in constructing a configuration–workload-aware performance model. Section 4 describes our proposed approach for building such a model. Section 5 introduces our setup method for evaluation. Section 6 presents the evaluation results of our approach in terms of accuracy and cost. Section 7 describes the SCPM algorithm and presents its corresponding evaluation results. Section 8 discusses the potential threats to validity, and Section 9 concludes the paper.

2 Background and Related Work

In this section, we introduce the background concepts and review prior research relevant to our study, focusing on performance modeling and queueing-based architecture models. These areas, together, provide the theoretical and methodological foundation for constructing models that can predict system behavior across different configurations and workloads. Specifically, we first discuss prior research on configuration-aware and workload-aware performance modeling, and then present how queueing-based models, such as queueing networks (QNs) and queueing Petri nets (QPNs), serve as a complementary approach for capturing system-level dynamics in software performance analysis. Finally, we introduce the tools and frameworks for microservice benchmarking.

2.1 Performance Modeling

Performance modeling aims to estimate system performance under different environmental and operational conditions without requiring exhaustive testing. Table 1 presents the purposes and contributions of prior research. Performance models have been widely applied in configuration optimization, capacity planning, performance regression detection, and performance prediction. In general, existing studies can be grouped into two major categories: configuration-aware performance modeling (see Section 2.1.1) and workload-aware performance modeling (see Section 2.1.2). However, configuration-aware and workload-aware performance modeling are not entirely independent research directions. Recent studies have explored hybrid approaches that combine configuration characteristics and workload behavior under specific scenarios (see Section 2.1.3).

2.1.1 Configuration-Aware Performance Modeling. Configuration-aware performance modeling, known as performance influence modeling, describes how configuration options and their interactions influence the performance of a configurable system [55]. The process of constructing these models is called configuration-aware performance modeling. Configuration-aware performance modeling helps reduce testing cost by learning a predictive model that generalizes performance outcomes across untested configurations. Such models have been widely used for various objectives, including minimizing testing efforts [7, 13, 16, 18, 20, 22, 26, 44, 54, 57, 64, 66], optimizing configuration settings [1, 17, 33, 45, 46], identifying configuration constraints [4, 49], and enabling model reuse across environments [14, 25, 30, 67].

Most prior work focuses on improving prediction accuracy and reducing the number of required samples. For example, Ha et al. [20], Cheng et al. [7], and Shu et al. [54] applied deep learning models such as feedforward neural networks to capture nonlinear interactions between configuration options. These deep learning methods achieve higher prediction precision than traditional approaches, such as linear regression [16, 18] or solver-based modeling [22], while requiring fewer samples. However, their accuracy evaluations rely on random sampling in most cases, which can lead to uneven coverage of performance features.

To improve sampling efficiency, Kaltenecker et al. [26], Xiang et al. [66], and Xia et al. [64] proposed distance-based or search-based sampling methods. These methods strategically select configurations that maximize the diversity of training samples, improving generalization with limited data. For instance, CoMSA [64] combines sampling with active learning to iteratively refine performance models based on intermediate training results, while Gong [13] utilized regression trees to cluster configurations and improve accuracy. Sun et al. [57] further enhanced the modeling process by automatically analyzing program syntax trees to identify potential configuration parameters at runtime.

Other research directions use configuration-aware models for optimization and constraint analysis. Oh et al. [46] applied statistical filtering to recursively select high-impact samples, while Nair et

al. [45] introduced a sequential model-based optimization method that searches for configurations with high predicted performance and low uncertainty. Chen et al. [4] employed random forest models to infer configuration rules and detect constraints that might lead to failures.

Recent studies focus on model evolution and transfer learning to reuse performance models across code changes or environments [30, 67]. Xiang et al. [67] proposed detecting performance drift to identify when retraining is necessary, while Krishna et al. [30] proposed BEETLE, a transfer-learning approach for reusing configuration-performance models across different environments. These approaches reduce retraining costs but typically assume stable workload conditions. When workload characteristics vary significantly, such configuration-aware models may no longer reflect real performance behavior.

In summary, configuration-aware performance modeling provides an effective way to reduce the cost of performance testing, identify optimal configuration options, and detect performance anomalies. However, these methods generally assume that workloads remain constant or follow a limited pattern. When both configurations and workloads change in modern software, the validity of existing configuration-aware models becomes limited.

2.1.2 Workload-Aware Performance Modeling. While configuration-aware models focus on system parameters, workload-aware performance modeling emphasizes how varying workloads influence performance. Such models are especially useful in load and stress testing, where developers analyze how systems behave under changing usage patterns, request mixes, or concurrency levels. A workload can be defined by user behavior, request type, or time-based arrival characteristics, allowing performance analysts to relate specific workload features to measurable outcomes such as latency or throughput.

Prior studies have proposed various strategies for capturing workload behavior. Shang et al. [52] clustered web application workloads to identify groups with similar performance trends, enabling efficient workload selection for testing. Ma et al. [40] constructed predictive models for database workloads by characterizing query patterns and execution profiles. Liao et al. [36] extended this idea to detect performance regressions across software versions by correlating workload characteristics with observed performance changes. Martin et al. [41] further enhanced workload-aware modeling by applying model evolution and transfer learning techniques to adapt performance models to workload shifts over time, improving prediction stability in evolving systems.

Workload-aware performance modeling has proven valuable for explaining how system performance varies with traffic composition or intensity, providing insights into potential bottlenecks and performance regressions. However, the configurations in workload-aware performance modeling methods remain static. This limitation makes the modeling method hard to reuse once the configuration has changed. As a result, while workload-aware models can effectively explain workload-driven performance variations, they are less suitable for scenarios that involve simultaneous changes in both workloads and configurations.

2.1.3 Hybrid performance modeling. Recent studies have explored hybrid performance modeling approaches under specific scenarios. Krishna et al. [30] construct workload-aware performance models under specific software versions or hardware configurations and then transfer the learned knowledge across different environments. Their work primarily focuses on investigating the feasibility of transfer learning across configurations, workloads, and software versions to reduce retraining cost. Franks et al. [12] mainly model workload intensity and architectural resource configurations to analyze contention and request propagation. Their work primarily focuses on system-level resource configurations, such as processor allocation, thread concurrency, and service deployment settings, rather than fine-grained software configuration options in configurable systems. Didona et al. [9] combined analytical queueing-based modeling with machine learning to

improve prediction robustness in configurable systems. Their approach uses analytical estimation as a baseline prediction and applies machine learning to model residual prediction errors under different configurations. Similarly, Comprex [60] combines structural information extracted from source code with local performance models to improve configurable-system performance prediction through modular aggregation. These hybrid approaches demonstrate that combining architectural or analytical knowledge with data-driven learning can improve prediction capability and robustness under limited training data.

Table 1. Comparison of related performance modeling studies.

Purposes	**Description**	**Config-aware**	**Workload-aware**	**Reference**
Reducing test effort	Reducing test effort by employing novel sampling methods that avoid redundant testing, or by using state-of-the-art modeling techniques to increase accuracy with a given sample size.	✓	✗	[7, 13, 16, 18, 20, 22, 26, 44, 54, 57, 60, 64, 66]
Optimizing configuration	Optimizing configuration settings and finding optimal configuration settings	✓	✗	[1, 17, 33, 45, 46]
Identifying configuration constraints	Identifying configuration constraints to avoid errors in systems.	✓	✗	[4, 49]
Detecting performance regression	Predict or forecast system performance to avoid performance regressions or error	✗	✓	[34, 40, 52]
Reusing models	Reusing performance model across environments to reduce cost of model training	✓	✗	[14, 25, 67]
	Reusing performance model in different environments to reduce cost of modeling	✗ (except env-related)	✓	[30, 41]
	Reusing performance model in different environments to detect performance regressions	✗	✓	[36, 37]
Improving performance prediction	Modeling and analyzing system performance using queueing-based models.	✗(except concurrency-related)	✓	[12]
	Improving performance prediction accuracy by integrating analytical models with machine learning.	✗(except env-related)	✓	[9]

2.2 Queueing Models for Software Performance

Queueing models are a foundational technique in software performance engineering for capturing concurrency, contention, and interactions among system components. A standard queueing network (QN) [27] models service centers connected by queues, allowing analytical estimation of throughput, response time, and resource utilization. Variants such as open, closed, and mixed networks enable modeling of different workload patterns and system boundaries. Layered Queueing Networks (LQNs) [12, 53] extend the basic QN formalism to represent layered architectures and nested service calls, making them well suited for multi-tier and microservice systems. LQNs support both analytical and simulation-based evaluation, allowing designers to examine how architectural or workload changes affect end-to-end performance. However, accurately building LQNs for large-scale microservice systems can be challenging due to dynamic workloads, complex service dependencies, and the need for precise runtime parameter estimation.

Queueing Petri Nets (QPNs) [28] integrate Petri nets with queueing theory, combining the causal and concurrent semantics of Petri nets with quantitative analysis from queueing networks. This hybrid modeling capability allows explicit representation of synchronization, stochastic behavior, and resource sharing. The Queueing Petri Net Modeling Environment (QPME) [29] supports graphical model construction and simulation, enabling analysts to parameterize models with empirical data. Recent studies [34, 37] show that QPN-based modeling effectively captures bottlenecks, queueing delays, and workload interactions in large-scale distributed systems.

Compared to data-driven methods, queueing-based approaches are easier to understand and offer high generalizability and reusability. They clearly describe how services depend on each other and how shared resources cause delays or bottlenecks. Queueing models can also simulate what happens when workloads increase or when configurations change, such as adding replicas or adjusting thread pools. These models help explain performance changes, identify bottlenecks, and support capacity planning. However, queueing-based approaches require either a grey-box or white-box environment that allows analysis of architectural knowledge, such as service dependencies. This limitation may prevent their use in situations where insufficient permissions are available to access the system.

In summary, queueing-based performance modeling, particularly using QPNs, provides a structured, interpretable foundation for analyzing system performance when comprehensive architectural knowledge is available. By integrating architectural representation with empirical data, queueing models can accurately predict how configuration and workload changes interact, forming a key basis for our proposed configuration–workload-aware modeling approach.

2.3 Microservice Benchmarking Generators

Microservice benchmarking frameworks provide standardized environments for evaluating the performance, scalability, and reliability of microservice-based systems. μBench [8] is a lightweight, configurable framework that enables researchers to construct synthetic microservice topologies with controllable workloads, resource constraints, and inter-service dependencies, thereby enabling fine-grained experimentation in performance modeling and testing. HydraGen [50] focuses on automatically generating microservice architectures with configurable dependency graphs, resource limits, and deployment constraints, enabling controlled exploration of system design spaces. MSTG (Microservice System Test Generator) [62] emphasizes workload-driven benchmarking by synthesizing realistic traffic patterns and service interaction sequences, facilitating end-to-end testing under diverse operational conditions. However, the limitations of HydraGen and MSTG are that they do not support external dependencies within the microservices in the system. In addition, ServiBench [51] is a novel meta-benchmarking tool for serverless applications in cloud

environments. The tool provides the framework that unifies deployment, workload generation, and metric collection for comparative benchmarking across microservice platforms, such as AWS.

In summary, HydraGen and MSTG automate microservice generation and workload synthesis but lack support for external dependencies, while ServiBench focuses on serverless environments. μBench is chosen in this study for its lightweight design and ability to model realistic microservice interactions under diverse workload and configuration settings.

3 Motivation

In this section, we primarily discuss the motivation behind our study. We also present an illustrative example to further explain our reasoning.

3.1 Challenge: Hybrid Model to Achieve Config–Workload Awareness

As shown in Table 1, most workload-aware performance models and configuration-aware performance models assume that workloads or configurations remain constant in real-world scenarios. In other words, prior works assume that the interaction or influence between workloads and configurations is constant in their scenarios. However, Mühlbauer et al. [43] demonstrate that influences exist between workloads and configurations, and they vary in scale. Lesoil et al. [32] study the precision of performance models for configurable systems (i.e., image processing tools) across different inputs (i.e., images). Their work illustrates how performance can vary under different workloads and configurations. Both studies demonstrate that different input images yield varying performance on the configurable systems, highlighting that the influence of workloads and configurations extends even within a single tool. Moreover, most performance modeling methods focus on testing tools or functions rather than system-level analysis. To truly understand the system's overall performance, a system-level analysis is necessary. Microservice performance depends not only on individual service times, but also on how requests propagate through service dependencies and accumulate in queues under different workloads. Although queueing models can effectively describe system architecture, prior research has not integrated queueing models of software services as constituent elements in system-level analysis. This makes it challenging to train an accurate performance model that considers both configuration and workload. The model must manage both the extensive combinations of workloads and configurations and their complex interactions.

Although several recent studies combine workload and configuration information, these approaches are typically designed for specific scenarios, such as transfer learning across environments, workload-driven architectural analysis, or resource-level configuration analysis. These works primarily focus on relatively stable scenarios, rather than large-scale configuration-workload interaction analysis in configurable systems. In these systems, local configuration changes may propagate non-linearly through dependent services, leading to workload-dependent performance behaviors such as spikes, saturation, or performance-equivalent regions. These behaviors are difficult to capture using purely end-to-end prediction models because workload propagation and queueing interactions may vary dynamically across service workflows. This challenge motivates the need for a topology-aware hybrid modeling approach that can explicitly represent service interactions and workload propagation at the system level.

3.2 A Motivating Scenario: Microservice Video Streaming System

To further clarify our motivation, consider a microservice-based video streaming system (Fig. 1). The system contains multiple services, such as transcoding, packaging, ad-insertion, and recommendation services. Different workloads may trigger different service workflows. For example, Workload 1 invokes the ad-insertion service after packaging, while Workload 2 invokes the recommendation service. Since configurable services, such as transcoding and encoding services, may

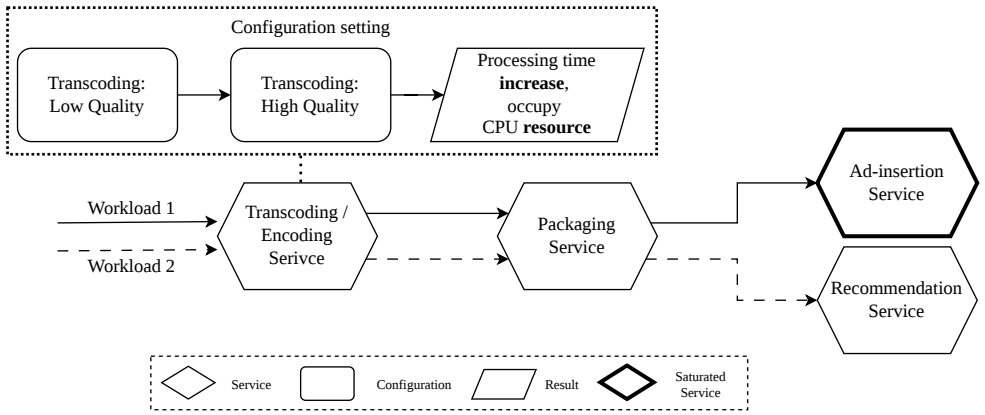


Fig. 1. Illustrative example of workload–configuration interactions in a microservice video streaming system.

have many runtime configuration options, testing all workload–configuration combinations before deployment is impractical.

A common practice is to evaluate only default configurations under typical workloads based on developer experience. However, this approach may miss performance risks caused by workload–configuration interactions after deployment. As shown in Fig. 1, when the transcoding configuration changes from low quality to high quality, the processing time of the transcoding/encoding service increases and consumes more CPU resources.

This configuration change may affect workloads differently because they follow different service paths. For Workload 2, the request continues from packaging to the recommendation service. In contrast, Workload 1 continues from packaging to the ad-insertion service, which may also require additional processing resources. As a result, the increased cost of transcoding can intensify resource contention along the Workload 1 path and eventually lead to saturation. This example illustrates that the impact of a configuration change cannot be fully understood without considering both workload behavior and system-level service interactions. If developers cannot identify potential failures or performance regressions in such scenarios, significant time and resources may be required to locate and resolve the issue after deployment.

To address this challenge, developers can construct workload-aware or configuration-aware performance models. Workload-aware models estimate bottlenecks under different traffic conditions, while configuration-aware models predict the performance impact of different configuration settings. However, these approaches often analyze workloads or configurations independently and may not accurately capture their combined influence on system-level performance.

Therefore, our goal is to design a hybrid model that integrates configuration-aware performance modeling with queueing-based architectural analysis. By modeling service-level behaviors and propagating workload interactions through the system architecture, the hybrid model can efficiently estimate performance risks under simultaneous workload and configuration changes while requiring significantly lower cost than exhaustive system-level testing.

4 Approach

In this section, we present HQP (**H**ybrid **Q**ueueing and Configuration-aware **P**erformance Modeling) to construct a hybrid performance model that considers both workloads and configurations. We

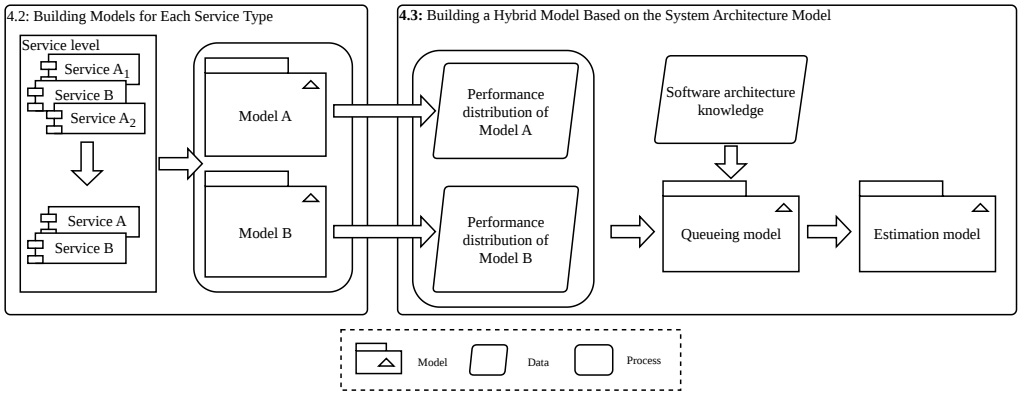


Fig. 2. An overview of HQP.

make use of the Queueing Petri Net formalism, complemented by a configuration performance modeling method. Figure 2 shows the overall process of the approach. We first construct performance models for individual services. Next, using software architectural knowledge, we integrate the performance models of individual services into the system architecture to construct a system-level performance model. We detail each step below.

4.1 Problem Statement

We assume that a modern software system can be divided into multiple modules based on their functions and logic. System behavior is influenced by dynamic workloads and configurations through queueing interactions. Based on this assumption, we focus on microservice systems, where service boundaries and inter-service interactions can be explicitly modeled.

We define services in the system as $S = (s_1, s_2, \ldots, s_n)$ and configuration options as $C = (c_1, c_2, \ldots, c_n)$ in each module. The configuration options may be numeric or categorical; each option c_i takes values from its domain $\mathcal{D}_i$, and a configuration is a vector $C = (c_1, c_2, \ldots, c_n) \in \mathcal{D}_1 \times \mathcal{D}_2 \cdots \times \mathcal{D}_n$. Let $\mathcal{C}$ be the domain of the configuration vectors $(C_1, C_2, \ldots, C_k)$. The workloads, which represent the URL request types, are denoted as $\mathcal{W} = (w_1, w_2, \ldots, w_m)$, while the workload frequencies are represented by $\mathcal{T} = (t_1, t_2, \ldots, t_m)$. The workload frequency represents the number of times each URL request type is sent per minute. For each workload type and corresponding workload frequency (e.g., w_1 has frequency t_1), under given configuration option settings, the corresponding performance measurements (i.e., processing time) are $\mathcal{Y} = (y_{(c_1,w_1,t_1)}, y_{(c_2,w_1,t_1)}, \ldots, y_{(c_n,w_m,t_m)})$. The independent variables can be $\mathcal{X} = \mathcal{S} \times \mathcal{C} \times \mathcal{W} \times \mathcal{T}$. Assuming the tested independent variable set is $\mathcal{X}_M$ and the corresponding performance measurement set is $\mathcal{Y}_M$. And, the unmeasured independent variables and performance measurements are $\{\mathcal{X}_U, \mathcal{Y}_U\}$.

Formally, the optimization problem is defined as follows:

$$\underset{\substack{\mathcal{X}_M \subseteq \mathcal{X} \\ |\mathcal{X}_M| = M}}{\arg\min} \frac{1}{|U|} \sum_{(x,y) \in U} \ell\left(f_{\mathcal{X}_M}(x),\, y\right) \tag{1}$$

where $f_{\mathcal{X}_M} : \mathcal{X} \to \mathcal{Y}$ denotes the model trained on the subset $(\mathcal{X}_M, \mathcal{Y}_M)$, and ℓ is loss function measuring the discrepancy between the prediction $f_{\mathcal{X}_M}(x)$ and the ground-truth performance y. Our goal is to build a highly accurate performance model with fewer samples, thereby making the modeling process more cost-efficient.

4.2 Building Models for Each Service Type

In prior studies, configuration-aware performance models [7, 20, 64] demonstrate high accuracy and cost efficiency when the configuration dimensionality is relatively small (about 5-20 configuration options). In a microservice system, applications are organized into service types, which may be reused or replicated to handle different workloads and configurations. As a result, it is unnecessary to construct separate configuration-aware performance models for every service. However, it is important to recognize that services can exhibit different behavior under varying workloads. This underscores that we cannot develop a simple model based solely on a small set of configurations for a specific workload. Therefore, for each service type, we build a configuration-aware performance model to analyze performance distribution. By focusing on individual services rather than the whole system, we can reduce both computational overhead and sample complexity while enabling scalable, reusable models across different system configurations.

In addition, for each service type, the configuration search space is relatively small, allowing the construction of an accurate model. Prior studies provide several configuration-aware performance modeling methods. In HQP, we apply CoMSA [64] to sampling and training the performance model at the service level. CoMSA is a sampling method that recursively selects samples based on the maximum prediction deviation across models trained on different subsets of the data. In each iteration, it identifies the sample with the greatest disagreement between models and adds it to the training set for the next round of testing. Based on CoMSA's empirical results, employing XGBoost as the modeling algorithm achieves high predictive accuracy while remaining cost-effective, even under a constrained sampling budget. By using CoMSA, we can ensure the performance model remains accurate and cost-efficient, which is crucial for large-scale microservice systems where efficiency and scalability are essential. Based on the COMSA results, when the sampling size reaches 5 times the number of configuration options, the accuracy of the prediction model can be guaranteed at a high level. We also evaluate the model accuracy in Section 5.

Moreover, building service-level configuration-aware models opens up opportunities for transfer learning, enabling knowledge from one service to be transferred to or adapted for similar services across different environments or versions. This further enhances our ability to rapidly and effectively model performance across diverse microservice architectures.

4.3 Building a Hybrid Model Based on the System Architecture Model

We apply the Queueing Petri Network (QPN) formalism in our approach to systematically describe and analyze the system architecture. Queueing Petri Nets provide a powerful framework for modeling both the structural and dynamic aspects of distributed software systems. QPN is particularly effective at modeling the impact of concurrent service requests, resource contention, and processing delays typical of modern microservice architectures. In our approach, the predicted processing times of individual services are integrated into the QPN to estimate how local configuration changes affect end-to-end system performance.

To construct the QPN model, we begin by mapping the major software services into corresponding places and transitions within the Petri net. Queueing places represent contention points where service requests may wait for processing, and transitions model the progression of jobs through service logic.

A critical aspect of setting up the QPN is parameterizing the queueing places with distribution parameters that reflect the stochastic nature of service times. Leveraging queueing theory [27], and in particular the memoryless nature of exponential distributions, each queueing place is modeled with an exponential service time distribution. This modeling choice is further supported by prior studies demonstrating that exponential service-time assumptions can reasonably approximate the

dynamics of real-world multi-tier systems [59]. Equation 2 defines the Probability Density Function (PDF) of the exponential distribution, which is commonly used in queueing theory to model the probability that the processing time of a request is less than or equal to t. This not only simplifies the queueing model but also aligns well with common models of real-world request processing.

$$f(t) = \lambda e^{-\lambda t}, \ t \geq 0 \tag{2}$$

To accurately reflect the performance of each module, the service rate λ is defined as the inverse of the mean processing time (m), i.e., $\lambda = 1/m$. Importantly, because we construct configuration-aware performance models for individual services, these empirical models provide accurate estimates of the processing time m across diverse configurations. For a given workload-configuration combination, the service-level performance model first predicts the mean processing time of each service under the corresponding workload and configuration settings. These predicted processing times are then converted into service rates and injected into the corresponding queueing places in the QPN model. The QPN subsequently models queueing contention and inter-service interactions across the microservice topology to estimate end-to-end system performance. Since we apply an exponential service-time distribution for queueing places, we assume that each service follows an exponential distribution under normal conditions. The processing time of individual services is therefore expected to follow an exponential distribution. Otherwise, the processing times estimated from individual services are not accurate for QPN-based analysis.

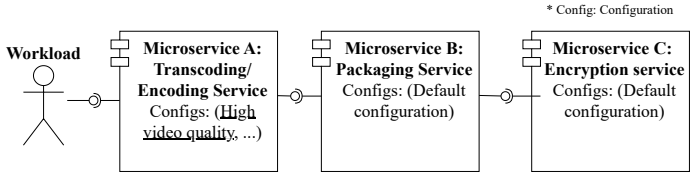
Once parameterized, we use the Queueing Petri net Modeling Environment (QPME) [28] to define, visualize, and refine the architecture-level QPN model. QPME allows us to intuitively represent the connectivity and processing characteristics of real applications. Figure 3a provides an illustration of the system's architecture, while Figure 3b shows the corresponding QPN instantiation. Workload frequencies and predicted service rates for each microservice are incorporated as model parameters, enabling simulation of realistic operating conditions. As a result, we can obtain a hybrid model that combines configuration-aware models of each service type and the QPN model.

After building the hybrid model, we employ SimQPN [29], a specialized tool for analyzing the models, to estimate end-to-end performance measures. By configuring the model with different workload types (e.g., specific API endpoints or URL request patterns) and system parameters, SimQPN can simulate the system's behavior and output average response times of services and workloads. The estimation results provide insights for identifying performance bottlenecks, supporting capacity planning, and evaluating the impact of configuration changes before deployment.

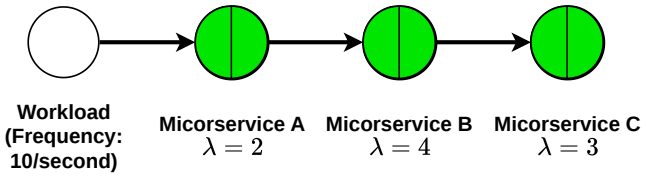
In summary, the hybrid model provides a practical and flexible way to evaluate and improve system performance at both the service and system levels as configurations and workloads change. By combining the queueing models of servers within a queueing Petri net that expresses the system architecture, we can develop an estimation model that accurately predicts end-to-end processing time across various configuration settings and workload types.

5 Experiment Setup

In this section, we primarily discuss the experimental setup. In our experiment, we require open-source systems that offer a large configuration space, support diverse and variable workloads, and allow for easy modification of configurations. These systems should also support various workload types in their operational scenarios. However, most qualified software systems, such as Netflix, are not open source. To simulate the real software architectures, we apply μBench [8] to build a dummy microservice software system. The μBench allows users to set up a software architecture and configure the functions used by each service in the system. Each service's functionality is implemented either by writing custom code or by invoking the tool's API. Therefore, to deploy



(a) Software architecture: an illustrative example.



(b) Queueing Petri net model based on software architecture. λ denotes the service rate.

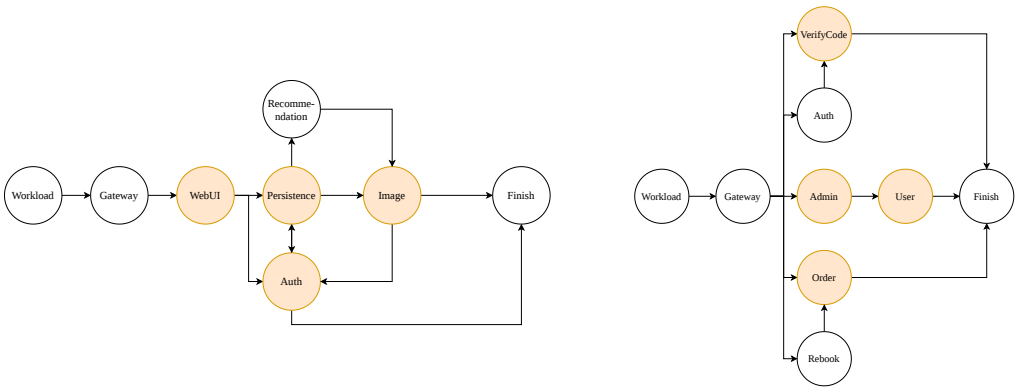
Fig. 3. An illustration example of characterizing software architecture features.

the microservice system using μBench, we must select an appropriate architecture and specify the tools for each service.

To ensure that our findings are not limited to simple experimental setups but can be generalized to more practical and diverse scenarios encountered in actual software deployments, we select one simple and one complex system architecture design: TrainTicket [69] and TeaStore [61], respectively. Both systems are widely used, real-world open-source architectures that have been adopted as benchmarks in prior research [19, 23, 37, 56, 58, 68]. The topology of the architectures is shown in Fig. 4. Each service follows a FIFO scheduling discipline. Compared with the TrainTicket architecture, TeaStore exhibits longer service invocation paths and more complex service dependencies. These architectural differences may lead to different queueing behaviors and workload propagation patterns. In addition, the evaluated systems include sequential and conditional service interactions, in which URL requests may traverse different service paths depending on workload behavior. To quantify their architectural complexity, we examine the average number of services called per workload: for the complex architecture, this average is 4.33, while for the simple architecture, it is 1.67. In actual runtime scenarios, the complex architecture continues to demonstrate greater complexity, averaging 4.24 services per workload compared to the simple architecture's 1.72. The results indicate that the two architectures have significant differences.

For the functions integrated into each service, we apply two types of relatively small but representative software components, FFmpeg[2] and lrzip[3], into our experimental environment. These components not only introduce configuration options and workload diversity but also simulate typical service-library dependencies in industrial deployments. These two tools are also widely used in prior works [20, 26, 43, 64]. In our actual modeling run in an isolated environment, FFmpeg achieved a mean relative error of 18.04% and a Spearman's rank correlation [11] of 0.93, while lrzip achieved a mean relative error of 1.49% and a Spearman's rank correlation of 0.98. As a result, we

[2]A complete, cross-platform solution to record, convert, and stream audio and video: https://ffmpeg.org/
[3]lrzip repository: https://github.com/ckolivas/lrzip



(a) Topology structure of TeaStore

(b) Topology structure of TrainTicket

Fig. 4. Topology structures of the evaluated microservice systems.

have two setups, which are the unified setup and the diverse setup. In the unified setup, we use only FFmpeg as the service function, whereas in the diverse setup, both FFmpeg and lrzip are used as service functions. The distinct features of these tools together provide comprehensive coverage of real-world scenarios.

After selecting the architectures and functions we need for deployment, we design four subject systems to achieve comprehensive coverage. The systems are designed as follows:

- TS-FF: Using TeaStore architecture as reference architecture with FFmpeg as the function only.
- TT-FF: Using TrainTicket architecture as reference architecture with FFmpeg as the function only.
- TS-FL: TeaStore architecture as reference architecture with both FFmpeg and lrzip as the function.
- TT-FL: TrainTicket architecture as reference architecture with both FFmpeg and lrzip as the function.

The colored services in Fig. 4a and 4b represent services that contain both FFmpeg and lrzip deployment in TS-FL and TS-FL. The execution path through these functions depends on the workload type. In contrast, the remaining services are shared infrastructure components and do not contain workload-dependent configurable deployments. This variety allows us to assess the effect of different component combinations and architectural compositions on system performance. Table 2 summarizes the details of the possible workload types and configurations in each subject. Different workload types follow different service queues within the system, resulting in distinct execution traces. Additionally, the configuration settings for each workload in each service are independent, further increasing the diversity and complexity of the workload space. The table highlights the exponential growth in complexity, making it impossible to test all combinations of the workloads and configurations.

Table 2. An overview of our subject systems.

Subject	Reference Architecture	Functions	$\lvert\mathcal{W}\rvert$	$\lvert C\rvert$	$\lvert\mathcal{S}\rvert$	$\lvert\mathcal{X}\rvert(\mathcal{T}=\mathcal{T}_0)$
TS-FF	TeaStore	FFmpeg	6	18	5	115200
TT-FF	TrainTicket	FFmpeg	6	18	6	46080
TS-FL	TeaStore	FFmpeg&lrzip	6	38	5	119296
TT-FL	TrainTicket	FFmpeg&lrzip	6	28	6	48640

$\lvert\mathcal{W}\rvert$:Total number of workload types. $\lvert C\rvert$: Total number of the configuration options. $\lvert\mathcal{S}\rvert$: Total number of services. $\lvert\mathcal{X}\rvert(\mathcal{T}=\mathcal{T}_0)$: Total number of combinations when workload frequencies is set as $\mathcal{T}_0$.

The μBench requires configuring a Kubernetes cluster[4] for deployment. Therefore, we create Kubernetes clusters using kubeadm[5] on three machines. Each machine is equipped with an Intel Core i7-12700 processor, 32 GB of RAM, and runs Ubuntu 22.04.5 LTS.

Workload generation is controlled using JMeter[6], which enables precise regulation and dynamic adjustment of HTTP/URL request frequencies during experiments. JMeter's flexibility allows us to create workload patterns that we generate in the fuzzing method. During workload generation, URL requests are randomly generated according to the specified workload frequencies.

6 Evaluation of HQP

Performance models are widely used for tasks such as predicting system performance, reducing testing time, and detecting performance regressions. However, if a performance model lacks sufficient accuracy, its predictions and diagnostic results become unreliable, limiting its practical value. In addition, if the cost of constructing and maintaining the model is prohibitively high, the model may be difficult to adopt in real-world scenarios. Therefore, we evaluate HQP from two perspectives: prediction accuracy and modeling cost.

To provide a comprehensive evaluation, we first compare HQP with state-of-the-art performance modeling baselines in terms of prediction accuracy and modeling cost. We then conduct ablation studies to assess the contributions of architectural knowledge and queueing-based modeling, thereby clarifying how each component affects the overall performance of HQP.

6.1 Evaluation of Modeling Accuracy

6.1.1 Method. In this part, we evaluate the prediction accuracy of HQP relative to state-of-the-art performance modeling techniques.

As discussed in Section 3, many existing performance modeling methods do not explicitly capture configuration–workload interactions. Therefore, we compare HQP with four representative configuration-aware performance modeling baselines: CoMSA [64] with XGBoost [6], CoMSA with DeepPerf [20], DaL [15], and HyPerf [10]. CoMSA with XGBoost provides a cost-effective and widely used tree-based baseline, while CoMSA with DeepPerf represents a deep-learning-based baseline with higher tuning overhead. DaL and HyPerf are included as state-of-the-art performance modeling baselines for configurable systems. We do not include BEETLE [30] and SeMPL [14] as baselines because their required learning setting is not available in our evaluation. Both methods require configuration-performance datasets collected from multiple source environments before transferring or generalizing the learned knowledge to a target environment.

To evaluate prediction accuracy, we use 10-fold cross-validation [21], since exhaustively measuring all configuration–workload combinations is infeasible. We use mean absolute relative error (MARE) to measure prediction error. Since HQP estimates service-level performance and then

[4]Cluster Architecture: https://kubernetes.io/docs/concepts/architecture/
[5]Kubeadm: https://kubernetes.io/docs/reference/setup-tools/kubeadm/
[6]JMeter documentation: https://jmeter.apache.org/index.html

propagates the results through the system architecture, we also evaluate ranking accuracy using Spearman's rank correlation coefficient [11]. Spearman's correlation allows us to assess whether the predicted performance values preserve the same relative ordering as the measured values. This is important for downstream tasks such as configuration selection and performance-spike detection, where preserving the relative ordering of configuration-workload settings can be as important as minimizing prediction error.

Following the sampling standard used by Kaltenecker et al. [26], we use the number of configuration options ($|C|$) as the basis for defining the sampling size n. Since HQP builds configuration-aware performance models for individual services, the sampling size of HQP corresponds to the number of service-level training samples. The system-level estimates are then obtained through QPME-based simulation using the learned service-level performance models.

6.1.2 Results. Table 3 presents the prediction accuracy of HQP and the baseline methods. Overall, HQP achieves stable prediction accuracy across subjects and sampling sizes, demonstrating both strong ranking accuracy and low prediction error. In several cases, HQP with a sampling size of n achieves accuracy comparable to or better than baseline methods using substantially larger sampling sizes, such as $5n$.

A special case occurs in TT-FL, where XGBoost with architectural knowledge achieves a slightly lower MARE than HQP when the sample size exceeds $3n$. This result is mainly due to the relatively simple reference architecture of TT-FL. As discussed in Section 5, this subject involves fewer related services per workload, with each workload using an average of 1.67 services. When architectural knowledge reduces the input space to a small set of highly related services, XGBoost can learn accurate mappings between input features and performance values.

In more complex systems, however, workloads may traverse more services and involve longer request traces. This increases the number of possible interactions between workload types, configuration options, and service dependencies, making purely data-driven prediction more difficult under limited sampling budgets. For example, TS-FF, which uses the more complex reference architecture, shows a larger gap between HQP and baselines without architectural knowledge than TT-FF. This result indicates that explicitly modeling service interactions is particularly beneficial when workloads propagate through more complex service topologies.

The results also show that architectural knowledge improves the performance of baseline methods. By incorporating service dependency information during preprocessing, XGBoost and DeepPerf achieve improved ranking accuracy and reduced MARE compared with their versions without architectural knowledge. However, these methods still do not explicitly model queueing effects or workload propagation. In contrast, HQP combines service-level performance prediction with QPN-based topology-aware propagation, enabling more accurate estimation of system-level performance.

Although DaL and HyPerf provide efficient baselines for configurable-system performance modeling, their prediction accuracy is less stable in our workload-configuration-aware microservice setting. DaL tends to perform better when the training data are sufficiently diverse, while HyPerf generally requires more samples to achieve high accuracy in its experiments. In contrast, HQP achieves stable prediction accuracy under limited sampling budgets.

Overall, the results demonstrate that HQP's accuracy comes from the combination of service-level performance modeling and queueing-based architectural propagation. The learned service models capture configuration-dependent performance variations, while the QPN captures system-level workload propagation and service interactions.

Table 3. Prediction accuracy with 10-fold cross-validation: Spearman's rank correlation (ρ) and mean absolute relative error (MARE) per subject.

Subject	Sampling Size	Model Performance (ρ/MARE)				
		HQP	XGBoost	DeepPerf	HyPerf	DaL
TS-FF	$n(18)$	**0.74**/29.69%	–/25.51%	–/**21.39%**	–/94.50%	–/47.53%
	$2n(36)$	**0.87/18.82%**	0.30/33.15%	0.09/30.88%	-4.21/69.76%	-4.31/45.65%
	$3n(54)$	**0.90/18.52%**	0.04/52.38%	0.16/75.23%	-1.46/53.85%	-1.08/41.98%
	$4n(72)$	**0.98/9.09%**	0.33/51.50%	0.25/73.42%	-0.85/44.48%	-1.04/45.64%
	$5n(90)$	**0.97/9.46%**	0.40/50.97%	0.43/32.48%	-0.85/48.13%	-0.90/45.93%
TT-FF	$n(18)$	**0.79/27.48%**	–/114.25%	–/49.09%	–/82.64%	–/66.35%
	$2n(36)$	**0.90/21.09%**	0.49/63.53%	0.55/68.66%	-5.98/87.73%	-18.62/64.29%
	$3n(54)$	**0.94/18.27%**	0.57/65.56%	0.64/62.73%	-1.93/163.97%	-1.97/62.12%
	$4n(72)$	**0.97/10.38%**	0.73/48.43%	0.69/72.49%	-1.59/134.58%	-1.08/53.94%
	$5n(90)$	**0.97/10.81%**	0.75/47.17%	0.78/68.20%	-2.22/139.16%	-0.68/59.81%
TS-FL	$n(38)$	**0.81/31.06%**	0.35/29.09%	0.40/31.48%	-1.39/52.76%	-4.90/36.09%
	$2n(76)$	**0.98/5.71%**	0.67/22.36%	0.71/30.91%	-0.76/48.60%	-0.31/31.17%
	$3n(114)$	**0.99/5.35%**	0.67/23.38%	0.72/27.21%	-0.16/60.05%	-0.08/28.50%
	$4n(152)$	**0.99/5.14%**	0.77/21.47%	0.78/19.57%	-0.53/63.21%	0.09/27.21%
	$5n(190)$	**0.99/5.37%**	0.78/20.49%	0.82/23.45%	-0.18/62.88%	0.19/28.15%
TT-FL	$n(28)$	**0.87/13.02%**	0.71/21.09%	0.62/37.74%	–/64.51%	–/30.28%
	$2n(56)$	**0.94/10.05%**	0.78/19.73%	0.75/37.37%	-1.50/60.43%	0.52/17.17%
	$3n(84)$	**0.94/9.22%**	0.84/17.98%	0.80/24.33%	-1.11/59.65%	0.58/14.21%
	$4n(112)$	**0.93/10.05%**	0.86/14.97%	0.84/30.86%	-0.72/53.81%	0.77/12.98%
	$5n(140)$	**0.93/9.85%**	0.91/13.50%	0.89/29.35%	-0.44/54.51%	0.79/12.89%

Notes. Each cell reports ρ/MARE. Bold marks the best per row (higher ρ, lower MARE). n denotes the number of configuration samples; "–" means insufficient data or cannot be calculated.

Table 4. The average cost of the performance models (normalized to 50 samples)

Methods	Setting time (s)	Testing time (s)	Training time (s)
HQP	< 1800	3953	349
XGBoost	$\ll 1$	14147	138
DeepPerf	4-225	12936	4066
HyPerf	$\ll 1$	1166	354
DaL	$\ll 1$	1340	28

6.2 Evaluation of Modeling Cost

6.2.1 Method. We measure modeling cost using three components: setup time, testing time, and training time. Setup time refers to the effort required to configure the model or prepare the required modeling artifacts. For models that do not use QPME, setup time is mainly determined by parameter tuning. For example, DeepPerf provides a tuning procedure to improve prediction performance, and we assume that this tuning step is performed at least once for each subject system. We measure

the total time cost under the $5n$ sampling-size setting (e.g., 90 samples in TS-FF), then normalize the measured cost to estimate the average time cost for 50 samples.

For HQP, setup time mainly comes from constructing the QPN model, including defining the architecture, configuring queueing places, specifying parameters, and preparing the QPME file. Since QPME provides a graphical modeling interface, this cost is primarily determined by the number of modeling elements and the required interactive operations. For example, constructing the complex reference architecture in QPME involves 26 queueing places, 57 connection elements, and 19 logic-related elements. Assuming an average operation time of 15 seconds per element, the total setup time is less than 1800 seconds. This setup cost is a one-time cost and can be reused across repeated experiments as long as the system architecture remains stable.

Testing time refers to the time required to collect performance measurements for the selected workload–configuration samples. Training time refers to the time required to train the predictive models after the measurements are collected. To compare costs fairly, we report the cost under sampling settings that achieve comparable prediction accuracy.

6.2.2 Results. Table 4 presents the average cost of the evaluated performance models. According to Table 3, HQP with a sampling size of n generally achieves prediction accuracy comparable to or better than baseline methods with substantially larger sampling sizes. Therefore, we compare the modeling costs under these sampling settings.

In terms of setup time, XGBoost, HyPerf, and DaL have low setup costs because they require little manual configuration. DeepPerf requires additional tuning overhead, resulting in a setup time ranging from 4 to 225 seconds, depending on the tuning strategy. HQP has a higher initial setup cost because it requires constructing the QPN model. However, this cost is reusable: once the QPN model is constructed, only minor updates are needed when the system architecture changes. Therefore, the one-time QPN construction cost can be amortized across repeated performance analysis and exploration tasks.

Regarding testing time, HQP requires substantially fewer performance measurements than traditional baselines because it achieves stable accuracy with smaller sampling sizes. This reduces the total testing cost, which is often the dominant cost in performance modeling. Although HyPerf and DaL exhibit lower testing and training costs in Table 4, their prediction accuracy remains less stable across subjects and sampling sizes, as shown in Table 3. In particular, these methods have difficulty maintaining reliable rank correlation under workload–configuration interaction scenarios.

Regarding training time, DeepPerf requires substantially longer training time in our environment because GPU acceleration is not used. With GPU support, the training-time gap between DeepPerf and other methods may decrease. In contrast, XGBoost, HyPerf, and DaL have relatively low training costs. HQP introduces moderate training overhead because it trains service-level performance models and performs QPN-based estimation, but this cost remains practical compared with the testing cost saved by reducing the number of required measurements.

Overall, HQP achieves a favorable trade-off between prediction accuracy and modeling cost. Although its initial setup cost is higher due to QPN construction, the model is reusable and provides stable topology-aware prediction accuracy under limited sampling budgets.

6.3 Evaluation of Contribution of Architectural Knowledge and Queueing Model

6.3.1 Method. To understand the contribution of architectural knowledge and queueing-based modeling, we conduct ablation studies that remove or isolate key components of HQP.

First, we evaluate a variant without QPN. In this setting, we remove the queueing-based topology propagation and directly use a regression model, i.e., XGBoost, to predict URL-level processing time

Table 5. Prediction accuracy with 10-fold cross-validation of ablation study: Spearman's rank correlation (ρ) and mean absolute relative error (MARE) per subject.

Subject	Sampling Size	Model Performance (ρ/MARE)				
		HQP	**QPN-only**	**Without-QPN**	**XGBoost w/ arch.**	**DeepPerf w/ arch.**
TS-FF	$n(18)$	**0.74**/29.69%	–/21.39%	–/46.44%	–/39.23%	–/27.14%
	$2n(36)$	**0.87/18.82%**	0.09/30.88%	-9.99/48.79%	0.61/40.17%	0.42/31.83%
	$3n(54)$	**0.90/18.52%**	0.16/75.23%	-2.35/42.59%	0.39/35.90%	0.32/40.38%
	$4n(72)$	**0.98/9.09%**	0.25/73.42%	-1.44/36.53%	0.65/32.54%	0.63/36.14%
	$5n(90)$	**0.97/9.46%**	0.43/32.48%	-0.18/32.08%	0.80/30.59%	0.68/28.32%
TT-FF	$n(18)$	**0.79/27.48%**	–/49.09%	–/27.99%	–/34.33%	–/49.68%
	$2n(36)$	**0.90/21.09%**	0.55/68.66%	-0.64/25.81%	0.66/35.19%	0.79/57.73%
	$3n(54)$	**0.94/18.27%**	0.64/62.73%	0.04/22.38%	0.78/30.79%	0.85/49.11%
	$4n(72)$	**0.97/10.38%**	0.69/72.49%	0.42/18.61%	0.81/30.15%	0.82/50.44%
	$5n(90)$	**0.97/10.81%**	0.78/68.20%	0.54/16.52%	0.85/24.43%	0.90/39.44%
TS-FL	$n(38)$	**0.81/31.06%**	0.40/31.48%	-0.57/25.66%	0.50/29.57%	0.62/25.10%
	$2n(76)$	**0.98/5.71%**	0.71/30.91%	0.36/18.41%	0.76/19.82%	0.78/18.74%
	$3n(114)$	**0.99/5.35%**	0.72/27.21%	0.56/14.87%	0.83/17.03%	0.83/20.97%
	$4n(152)$	**0.99/5.14%**	0.78/19.57%	0.68/12.68%	0.87/16.01%	0.85/19.07%
	$5n(190)$	**0.99/5.37%**	0.82/23.45%	0.76/10.79%	0.89/14.77%	0.90/15.80%
TT-FL	$n(28)$	**0.87/13.02%**	0.62/37.74%	-/29.18%	0.77/13.84%	0.81/22.62%
	$2n(56)$	**0.94/10.05%**	0.75/37.37%	-0.31/23.57%	0.87/14.30%	0.77/19.42%
	$3n(84)$	**0.94/9.22%**	0.80/24.33%	0.65/17.18%	0.90/11.15%	0.91/25.87%
	$4n(112)$	**0.93**/10.05%	0.84/30.86%	0.77/14.47%	0.90/**10.01%**	0.94/12.61%
	$5n(140)$	**0.93**/9.85%	0.89/29.35%	0.78/12.44%	0.92/**9.74%**	0.95/19.70%

Notes. Each cell reports ρ/MARE. Bold marks the best per row (higher ρ, lower MARE). n denotes the number of configuration samples; "–" means insufficient data or cannot be calculated, "w/arch." indicates that the method utilizes architectural knowledge during preprocessing.

from the predicted processing times of individual services. This variant allows us to assess whether service-level architectural information alone is sufficient for accurate system-level prediction.

Second, we evaluate a QPN-only variant. Since QPN alone cannot capture configuration-dependent service performance variations, we use the processing time under the default configuration and apply workload frequency information in the QPN. This setting allows us to evaluate the contribution of queueing-based architectural propagation without learned service-level performance models.

Third, we evaluate architecture-aware baseline models. In this setting, baseline models use service dependency information during preprocessing, allowing us to assess whether architectural knowledge improves prediction accuracy even without explicit queueing-based propagation. We select XGBoost and DeepPerf as representative baselines for this ablation because they show comparatively strong performance among the evaluated methods.

6.3.2 Results. Table 5 presents the results of the ablation experiments. Overall, HQP achieves the best prediction performance across most subjects and sampling sizes, demonstrating the effectiveness of jointly using architectural knowledge and queueing-based modeling.

The QPN-only variant generally exhibits poor prediction accuracy and unstable ranking performance. Although the queueing model captures structural relationships among services, it cannot model configuration-dependent service performance variations without learned service-level performance models. As a result, its estimates are overly simplified and lead to weak ranking capability and high prediction error.

The Without-QPN variant performs better than the QPN-only setting, indicating that service-level architectural information contributes to prediction accuracy even without explicit queueing analysis. However, its performance remains consistently lower than HQP, particularly in ranking accuracy. This result indicates that QPN provides topology-aware performance propagation information that cannot be fully captured by direct regression-based aggregation.

To evaluate the contribution of architectural knowledge, we select two of the best baseline methods(i.e., XGBoost and DeepPerf) from Section 6.1 to generate the architecture-aware baselines. The architecture-aware baseline methods, namely XGBoost w/arch. and DeepPerf w/arch., generally improve over their corresponding versions without architectural preprocessing, demonstrating the usefulness of incorporating service dependency information. Nevertheless, they still underperform compared with HQP in most cases. This suggests that architectural knowledge alone is helpful but insufficient; explicitly modeling workload propagation and queueing interactions further improves prediction accuracy.

Overall, the ablation study results demonstrate that both architectural knowledge and queueing-based modeling are necessary components of HQP. Service-level performance models capture configuration-dependent behavior, while QPN enables topology-aware propagation across microservice workflows. The combination of these two components allows HQP to achieve a more accurate and stable prediction than either component alone.

Summary of Findings

HQP achieves stable prediction accuracy with substantially fewer measured samples than several baseline methods. It maintains high rank correlation and low prediction error by combining service-level performance modeling with QPN-based topology-aware propagation. Architectural knowledge improves the accuracy of baseline models, while queueing-based modeling further improves HQP's ability to capture workload propagation and service interactions. Overall, HQP provides a practical balance between prediction accuracy and modeling cost under limited sampling budgets. The ablation study further confirms that both architectural knowledge and queueing-based modeling are necessary to achieve HQP's stable prediction performance.

7 Exploration on Searching Performance Condition with HQP

Since HQP delivers strong results in both accuracy and cost, we aim not only to provide an estimation model that passively forecasts performance for given parameters, but also to proactively identify potential performance conditions that may arise as configurations or workloads change. In prior studies, detecting or forecasting performance spikes and saturation conditions often requires collecting and analyzing raw data from extensive performance testing. Reducing the cost of performance modeling requires avoiding training on redundant samples. As shown in Table 6, the primary goal of previous studies has been to avoid or detect these potential performance conditions. This process is both time-consuming and resource-intensive since all data is built on tested samples. Since our approach offers a flexible and cost-effective solution that can accurately estimate system

performance without real running, we propose SCPM (**S**earch and **C**ondition **P**rediction on hybrid **M**odeling) for forecasting the potential performance conditions.

7.1 SCPM

SCPM is a search-based method based on PerfFuzz [31], designed to systematically explore both configuration and workload spaces in microservice systems to identify potential performance risks. In PerfFuzz, a new test case is generated by mutating the values of an existing seed. Such a process continues until code coverage no longer increases, meaning no new behaviors are exercised by further mutations. Except for using code-coverage-inspired exploration to determine whether mutated cases provide new coverage, we abstract the behavioral status of the services to identify previously unseen system states. Specifically, the status of services consists of two components: (i) the rank ordering of the estimated mean service times across services and (ii) the saturation status of the services. For example, the original service status may be represented as [(3, 1, 4, 2), ()], where (3, 1, 4, 2) denotes the rank ordering of the estimated mean service times, and () indicates that no service is saturated. After mutation, the service status may change to [(3, 1, 4, 2), (3)], where (3) indicates that service 3 is saturated. If a newly generated status has not appeared in the historical status set, it is treated as new coverage and added to the history set. However, assessing test case coverage becomes challenging when both configuration and workload variations are considered [39]. Although sampling methods are effective at extending coverage across a wide range of options, they generally cannot cover all systems' actual performance behavior under these options. As a result, we manually set the limit of test case generation, except for calculating it based on coverage.

Furthermore, traditional testing covers only a narrow set of system conditions, but varied configurations and workloads in real deployments can trigger edge-case scenarios that are easily overlooked. SCPM leverages randomness and probabilistic sampling to generate test cases that not only broadly cover configuration options but also identify critical cases.

A key advantage of SCPM is its integration within our queueing model-based simulation environment. This integration enables us to estimate the performance impact of each generated test case through simulation, significantly reducing the need for costly, time-consuming real-world testing. Specifically, SCPM relies on feedback from SimQPN reports to iteratively focus its search on areas near the performance boundaries.

In conclusion, SCPM provides a powerful and efficient approach for identifying performance risks in complex microservice environments. The Algorithm 1 outlines the mechanism of our fuzzing approach as follows:

- **Lines 1-2:** Set up a random-workload firing weight to simulate fuzzing under randomized conditions.
- **Lines 3-6:** Build the probability list by collecting available modules, configurations, and candidate values.
- **Lines 7-8:** Initialize the three-dimensional probability list $Prob3d$ to record the probability of applying combinations of configurations and workloads. Set the child-count limit (e.g., $m = 100$).
- **Lines 9-10:** For each workload frequency, generate an initial seed based on $Prob3d$.
- **Lines 11-13:** For each seed in $\mathcal{P}$, mutate the seeds to generate children.
- **Line 14:** Run child to generate SimQPN report.
- **Lines 15-16:** Based on the report, we can obtain the feedback from the status of each module. If the new status of each module, add the seed to $\mathcal{P}$.
- **Line 18:** Update $Prob3d$ based on the feedback of modules.

Algorithm 1 The SCPM algorithm

```
1: 𝒲 ← {w_0, w_1, ..., w_i}                          ▷ Workload type set
2: 𝒯 ← Random({t_1, t_2, ..., t_n}, 𝒲)               ▷ Workload frequency set
3: 𝒮 ← {s_0, s_1, ..., s_j}                          ▷ Service type set
4: 𝒞 ← GetAvailiableConfigOption(𝒮)
5: I, J ← length(𝒲), length(𝒮)
6: Prob3d ← Fill3D(I, J, C, 1)
7: NumChildren ← 100
8: for t in 𝒯 do
9:     𝒫 ← {GenerateSeed(Prob3d)}
10:    for seed in 𝒫 do
11:        i ← 0
12:        for 0 ≤ i ≤ NumChildren do
13:            child ← Mutate(seed)
14:            feedback ← Run(child)
15:            if NewCase(feedback) then
16:                𝒫 ← 𝒫 ∪ {child}
17:            end if
18:            Prob3d ← UpdateProb3d(Prob3d, feedback)
19:            i ← i + 1
20:        end for
21:    end for
22: end for
```

7.2 Evaluation on SCPM

By introducing a search-based forecasting method, we can significantly reduce testing costs if the forecast is sufficiently accurate. Therefore, our goal is to evaluate how accurately SCPM can detect key performance conditions, specifically: performance-equivalent, performance-saturated, and performance-spiked conditions.

7.2.1 Method. In this research question, we aim to investigate how to identify potential performance conditions of the system. Before initiating the search process, it is essential to clarify which performance conditions are important to developers. Drawing from the objectives of previous studies, as summarized in Table 2, we identify five main purposes for performance modeling. From these, we summarize three performance conditions that are particularly important to developers: performance-equivalent, performance-saturated, and performance-spiked conditions. Table 6 summarizes the definitions and calculation methods for each condition. To provide a clear explanation, Table 7 lists all the parameter settings and the numerical values of the parameters.

To validate the approach, we measure the accuracy of our method's estimation for each performance condition. Table 10 reports the estimation accuracy of the estimation of SCPM on different performance conditions. A performance-equivalent condition is defined as a configuration/workload change in services that does not alter the request's processing time. To verify this condition, we compare the processing time before and after the changes. A performance-spiked condition represents a sudden increase in performance triggered by a one-time configuration/workload change. We apply the z-score algorithm[7] to detect the spikes. The performance-saturated condition occurs when the system reaches its bottleneck, potentially leading to errors.

[7]Find spikes in a dataset: https://rstudio-pubs-static.s3.amazonaws.com/640899_e805255c62a04dd8b3c8440329b07b40.html

Table 6. Definition of potential performance conditions.

Potential performance condition	Motivation	Requirement	Related purpose
Performance-equivalent	Small configuration/workload changes may yield statistically indistinguishable performance; grouping these cases avoids chasing noise and allows us to focus on impactful factors.	$\Delta(\mathbf{x_{t_1}}, \mathbf{x_{t_2}}) = \frac{\lvert x_{t_1} - x_{t_2} \rvert}{\max(\lvert x_{t_2} \rvert, \delta)} \leq \varepsilon$ where $\mathbf{x_{t_1}}$ and $\mathbf{x_{t_2}}$ are the performance vectors of two different configuration–workload settings; δ is a small constant to avoid division by zero. If the relative difference is smaller than ε (e.g., 0.05), the two settings are performance-equivalent.	Reducing test effort, Reusing model [7, 13, 14, 16, 18, 20, 22, 25, 26, 30, 41, 44, 54, 57, 64, 66, 67]
Performance-spiked	A subset of changes causes short-lived bursts (e.g., elevated tail processing time) without a steady-state shift; we flag these as spikes to aid triage.	$\mu_t = \frac{1}{2L+1} \sum_{j=-L}^{L} x_{t+j}$ $\sigma_t = \sqrt{\frac{1}{2L}} \sqrt{\sum_{j=-L}^{L} (x_{t+j} - \mu_t)^2}$ $z_t = \frac{x_t - \mu_t}{\max(\sigma_t, \delta)}$ $\text{spike}_t = \mathbf{1}\left[\max_{t=1,\ldots} z_t \geq \tau\right]$ where x_t at time t, L is half- window size (before and after t excluding t), τ is a z-score threshold (e.g., 3), and δ is a small constant.	Optimizing configuration, Identifying configuration constraints, Detecting performance regressions, Reusing models [1, 4, 17, 33, 34, 36, 37, 40, 45, 46, 49, 52]
Performance-saturated	Steady-state resource exhaustion leads to a throughput plateau while latency and/or error rate increase; identifying saturation enables safe early stopping and capacity planning.	$\text{satur}_t = \mathbf{1}\left[y \leq (1-\beta)\, s \vee \ e \geq \eta\right]$ where y represents the observed throughput, s is the configured frequency, and e denotes the error rate during testing. By setting a tolerance parameter $\beta \in [0, 1)$ (e.g., 0.10) and an error threshold η (e.g., any error if $\eta = 0$), we can more accurately determine when the system is considered saturated or at risk of performance issues.	Optimizing configuration, Identifying configuration constraints, Detecting performance regressions, Reusing models [1, 4, 17, 33, 36, 37, 45, 46, 49]

To verify estimation accuracy, we apply QPME parameters to the actual configuration settings in μBench and the workload settings in JMeter. If the real system performance matches the requirements of the performance condition under test, we consider the simulation result to be accurate in our estimation.

Table 7. Key parameters used in defining performance conditions.

Parameter	Meaning	Value
x	Processing time under a specific configuration–workload setting.	$(0, +\infty)$
t	Timestamp used for evaluating potential performance conditions.	*
ϵ	Threshold for determining whether two cases are performance-equivalent.	0.05
δ	Small constant to avoid division by zero during normalization.	1×10^{-9}
L	Half-window size for computing local statistics in spike detection.	3
τ	Z-score threshold for identifying transient performance spikes.	3
β	Allowed relative throughput degradation before declaring saturation.	0
η	Error-rate threshold used in saturation detection.	0
s	Configured request frequency (requests per second).	$[10, 70]$
e	Observed error rate during testing.	$[0, 1]$

* The value of t depends on the number and sequence of evaluated cases.

As shown in Table 10, we set the z-score threshold based on the recommendation in practice[7]. To calculate two performance results that are equivalent, based on the mean deviation of the repeat runs (i.e., 0.10 in the complex reference architecture and 0.05 in the simple reference architecture), and 0.10 is an acceptable parameter to avoid the influence of system noise. Using JMeter to verify the condition, we conclude two reactions of the system under the performance-saturated condition. One obvious system behavior is that it reports time-out errors for URL requests, which is the default setting of 300 seconds. However, to detect more performance-saturation cases, we set the timeout to 100 seconds. Another standard is that the system throughput is lower than the configured URL request rate.

To evaluate the search effectiveness of SCPM, we compare our approach against two baselines. First, we replace the guided fuzzing strategy in SCPM with purely random selection to assess the contribution of the proposed exploration strategy. In this comparison, we primarily analyze the number of detected potential performance conditions under SCPM and random selection.

Second, we compare SCPM with SMAC (Sequential Model-based Algorithm Configuration) [24], a widely used model-based optimization technique for parameter tuning and configuration search [5]. In this part, we focus on both the detection effectiveness and exploration cost of SCPM and SMAC-based methods.

To provide a more comprehensive comparison, we implement two objective-guided variants of SMAC with different search priorities: SME and SMS. SME (SMAC for performance-equivalent conditions) is designed to prioritize the discovery of performance-equivalent conditions. Its optimization objective favors configurations whose performance behaviors are similar to previously observed cases, enabling the search process to efficiently identify regions with comparable performance characteristics. In contrast, SMS (SMAC for performance-spiked conditions) focuses on identifying performance-spiked conditions. SMS biases the exploration toward configurations exhibiting significant local deviations or abnormal performance behaviors, thereby increasing the likelihood of discovering transient spikes and unstable regions in the configuration space.

In addition, the scoring mechanism used in SMS is closely related to the criterion for identifying performance-saturated conditions, as both emphasize abnormal degradation and throughput-related

behaviors. Therefore, instead of introducing a separate saturation-oriented SMAC variant, we integrate the detection of performance-saturated conditions into SMS. This design reduces redundancy in the optimization objectives while enabling SMS to explore two performance conditions.

7.2.2 Results. Table 8 presents the number of detected performance conditions per 100 trials for the four evaluated methods. Overall, SCPM consistently outperforms the random selection strategy across different types of performance conditions, demonstrating the effectiveness of guided exploration. The results further show that SME achieves the highest overall number of detections across most performance conditions and subjects, indicating that equivalent-oriented search is effective for discovering stable performance behaviors.

SMS identifies a larger number of performance-saturated conditions, as its scoring mechanism emphasizes abnormal degradation and saturation-related behaviors. However, SMS does not consistently outperform SME in detecting performance-spiked conditions. This is primarily because performance-spiked conditions are relatively rare and localized in the search space, making them more difficult to discover reliably even with spike-oriented search objectives.

Nevertheless, both SME and SMS require iterative sample evaluation and model updating during the search process, resulting in higher computational and testing overhead. Table 9 reports the cost of the four methods for every 100 trials. As expected, the random selection strategy incurs the lowest overhead because it does not require additional scoring or optimization procedures. In contrast, the SMAC-based methods introduce substantial tuning costs due to repeated model construction, scoring, and sample selection during exploration.

Considering both exploration efficiency and search cost, SCPM achieves a favorable balance between detection capability and overhead. Although SCPM may not always achieve the highest detection count, it can efficiently discover diverse performance conditions while maintaining a significantly lower exploration cost than the SMAC-based approaches. However, in simpler architectures, SCPM detects fewer performance conditions than SMAC-based methods, which narrows its practical advantage despite its substantial reduction in computational cost.

Table 10 reports the accuracy achieved in estimating the three categories of performance conditions. As the results indicate, SCPM consistently maintains accuracy above 80% across most conditions, demonstrating the robustness of our method.

Notably, in the TS-FL system, the estimation accuracy in performance-spiked conditions is lower than in other conditions, due to the limited number of such conditions. Through closer examination, we attribute these inaccuracies primarily to the occasional overestimation by individual service performance models. Even though these models generally maintain high configuration prediction accuracy, rare outliers may propagate through and influence the synthesized QPME system-level estimates.

Further analysis, as shown in Table 3, indicates that the diverse service function setup increases prediction accuracy because the newly joined subject is highly predictable. This feature also indicates that the diverse service function setup exhibits a more uniform performance distribution, as the occurrence of performance-spiked conditions in TS-FL and TT-FL is lower than in TS-FF and TT-FF, respectively. This demonstrates the effect of component-level diversity on the difficulty of performance modeling, suggesting that building stable and predictable services can enhance the overall stability of the system.

In Table 10, the performance-equivalent and performance-saturated conditions in the TS-FF and TS-FL systems are higher than those in the TT-FF and TT-FL systems, respectively. This is related to the workload/user profile we applied to the system. As shown in Section 5, the architecture's complexity differs across experiments due to the workload profile we applied. The complexity of the architecture can increase the number of performance-equivalent and performance-saturated

Table 8. Number of detected performance conditions per 100 samples under different methods.

Subject	Performance-equivalent				Performance-spiked				Performance-saturated			
	R	SME	SMS	SCPM	R	SME	SMS	SCPM	R	SME	SMS	SCPM
TS-FF	0.00	12.43	0.00	4.29	0.00	16.43	2.86	6.29	70.57	75.71	86.14	75.00
TT-FF	0.00	7.29	0.00	0.86	0.00	14.00	4.57	4.86	26.29	55.86	86.57	69.57
TS-FL	0.00	17.43	1.29	3.57	0.00	17.14	4.43	3.29	78.43	82.71	53.43	80.71
TT-FL	0.00	9.71	3.43	2.29	0.00	16.71	2.14	6.71	28.57	62.71	70.57	21.29

Notes. R denotes the random selection method, while SME denotes the equivalent-oriented SMAC method, and SMS denotes the spike-oriented SMAC variant.

conditions, as complex reference architectures have more features that introduce more interactions between services and configuration options.

We also observe that the performance-saturated condition achieves higher accuracy compared to the other two performance conditions. This is because the requirement for identifying the performance-saturated condition is less strict. In high workload frequency scenarios, most test cases easily meet the saturation criterion, as the system is more likely to reach saturation. Consequently, the high proportion of correctly identified cases in this category increases the overall accuracy. For example, in the TS-FF system, there are 316 test cases with a workload frequency below 40 per minute; of these, 251 are accurate, resulting in an accuracy rate of 79.43%. Since the SCPM is designed to find test cases near saturation, our test cases are less likely to mutate saturated cases, which decreases the probability of detecting outliers in high workload frequency scenarios. This observation suggests that search-based strategies for high and low workload frequency scenarios should differ in their test case generation approaches, with greater emphasis on leveraging system knowledge in high workload frequency environments.

Table 9. The average cost of the performance models every 100 trials.

Methods (per 100 trials)	Setting time (s)	Testing time (s)	Training time (s)
R	-	574	-
SME	-	32981	-
SMS	-	33204	-
SCPM	<1800*	593 + 3953*	349*

*Notes. R denotes the random selection method, while SME denotes the equivalent-oriented SMAC method, and SMS denotes the spike-oriented SMAC variant. * represents a one-time investment for building a performance model using HQP.*

7.3 Sensitivity of Parameters

In this section, we analyze the sensitivity of the parameters listed in Table 7. As certain parameter modifications can exponentially increase the number of detectable performance conditions, exhaustively evaluating all conditions is infeasible. Therefore, we evaluate a representative subset of conditions. Specifically, we select 100 samples from the total candidates that do not overlap with those identified under the default parameter settings. To ensure diversity, we adopt a stratified sampling strategy over configuration and workload options. For example, we include samples with distinct configuration settings (e.g., one using the '-r' option and another using the '-l' option in the s0 pod).

Table 10. The accuracy of our approach in estimating performance conditions.

Subject	Accuracy of detection on potential performance conditions		
	Performance-equivalent	Performance-spiked	Performance-saturated
TS-FF	(245/268) 91.42%	(25/31) 80.65%	(668/727) 91.80%
TT-FF	(5/6) 83.33%	(137/159) 86.16%	(486/487) 99.79%
TS-FL	(270/309) 87.38%	(8/11) 72.73%	(1258/1290) 97.52%
TT-FL	(13/16) 81.25%	(47/47) 100%	(139/149) 93.29%

Notes. Values in parentheses denote the number of correctly detected cases over the total cases.

Table 11. Sensitivity analysis of threshold ϵ

Subject	Precision of detection on potential performance-equivalence conditions		
	$\epsilon = 0.05$ (default)	$\epsilon = 0.10$	$\epsilon = 0.15$
TS-FF	245/268 (91.42%, N=268)	14/100 (14.00%, N=1746)	12/100 (12.00%, N=2116)
TT-FF	5/6 (83.33%, N=6)	32/100 (32.00%, N=985)	13/100 (13.00%, N=2116)
TS-FL	270/309 (87.38%, N=309)	23/100 (23.00%, N=882)	6/100 (6.00%, N=1772)
TT-FL	13/16 (81.25%, N=16)	26/82 (31.71%, N=82)	18/100 (18.00%, N=208)

Notes. Values are reported as correctly detected cases / manually evaluated samples (precision, total candidate pool size N).

7.3.1 Sensitivity of Parameters Related to Performance-Equivalent Conditions. In this section, we analyze the sensitivity of the parameter used to classify performance-equivalent cases, namely the threshold ϵ.

The threshold ϵ determines whether the performance difference between two cases is small enough to be considered equivalent. A larger ϵ relaxes this criterion, allowing more cases with larger performance differences to be classified as equivalent. Consequently, increasing ϵ leads to more detected performance-equivalent cases.

However, this relaxation also increases the likelihood of false positives, as cases that are not truly equivalent may be incorrectly classified as such. To balance detection coverage and accuracy, SCPM sets $\epsilon = 0.05$ as a conservative default.

To further understand the impact of ϵ, we evaluate its sensitivity at $\epsilon = 0.05$, 0.10, and 0.15. For $\epsilon = 0.10$ and 0.15, we focus on cases predicted as performance-equivalent by HQP but not identified under the default setting, to examine potential misclassifications.

Table 11 reports the precision of detection. As shown in the table, increasing ϵ significantly increases the number of detected conditions (i.e., expands coverage), but at the cost of reduced precision. This result highlights a clear trade-off: a smaller ϵ yields more reliable detections, while a larger ϵ improves coverage but introduces more false positives.

7.3.2 Sensitivity of Parameters Related to Performance-spiked Conditions. Two parameters affect the detection of performance-spiked cases: the window size L and the z-score threshold τ.

The parameter L defines the window size used for spike detection. Specifically, when $L = 3$, each candidate point is evaluated using a local window consisting of three preceding and three succeeding cases. The choice of L reflects a trade-off between sensitivity and robustness: a smaller L

Table 12. Sensitivity analysis of threshold L.

Subject	Precision of detection on potential performance-spiked conditions		
	$L = 1$	$L = 3$ (default)	$L = 5$
TS-FF	0/100 (0.00%, N=547)	25/31 (80.65%, N=31)	20/25 (80.00%, N=25)
TT-FF	0/100 (0.00%, N=706)	137/159 (86.16%, N=159)	33/36 (91.67%, N=36)
TS-FL	0/100 (0.00%, N=491)	8/11 (72.73%, N=11)	4/7 (57.14%, N=7)
TT-FL	0/100 (0.00%, N=375)	47/47 (100.00%, N=47)	8/8 (100.00%, N=8)

Notes. Values are reported as correctly detected cases / manually evaluated samples (precision, total candidate pool size N).

Table 13. Sensitivity analysis of threshold τ.

Subject	Precision of detection on potential performance-spiked conditions		
	$\tau = 1$	$\tau = 3$ (default)	$\tau = 5$
TS-FF	55/100 (55.00%, N=1014)	25/31 (80.65%, N=31)	7/7 (100.00%, N=7)
TT-FF	10/100 (10.00%, N=985)	137/159 (86.16%, N=159)	28/36 (77.78%, N=36)
TS-FL	50/100 (50.00%, N=882)	8/11 (72.73%, N=11)	5/6 (83.33%, N=6)
TT-FL	17/100 (17.00%, N=508)	47/47 (100.00%, N=47)	8/8 (100.00%, N=8)

Notes. Values are reported as correctly detected cases / manually evaluated samples (precision, total candidate pool size N).

makes the detection more sensitive to local fluctuations, while a larger L smooths the local statistics and reduces noise, but may overlook sharp spikes.

Table 12 presents the sensitivity results for different values of L. When $L = 1$, no reliable spikes are detected across all subjects, indicating that the window is too small to establish a stable local baseline for spike identification. Increasing L to 3 significantly improves detection precision, as the larger context enables more reliable identification of abnormal deviations. When L is further increased to 5, the precision remains comparable or slightly decreases in some cases, suggesting that overly large windows may smooth out local variations and reduce sensitivity to sharp spikes. Therefore, $L = 3$ provides a balanced choice between sensitivity and robustness.

The parameter τ defines the z-score threshold for determining the significance of a spike. A smaller τ lowers the detection threshold, resulting in more detected spikes, while a larger τ imposes a stricter criterion and detects only highly significant deviations. We adopt $\tau = 3$ as the default value, following the classical three-sigma rule [8].

Table 13 shows the sensitivity of τ. When $\tau = 1$, a large number of cases are classified as spikes, but the precision is relatively low, indicating a high rate of false positives. As τ increases to 3, the precision improves substantially across all subjects, demonstrating a better balance between detection coverage and accuracy. When $\tau = 5$, the precision is high, but the number of detected cases is significantly reduced, indicating that only extreme spikes are captured while moderate spikes may be missed.

Overall, these results reveal a clear trade-off: smaller τ increases coverage but reduces precision, while larger τ improves precision at the cost of missing relevant spikes. Based on this observation, $\tau = 3$ is selected as a balanced default setting.

[8]NIST/SEMATECH e-Handbook of Statistical Methods: https://www.itl.nist.gov/div898/handbook/toolaids/pff/pmc.pdf

Table 14. Sensitivity analysis of timeout setting.

Subject	Precision of detection on potential performance-saturated conditions	
	timeout=100s (default)	timeout=300s
TS-FF	668/727 (91.80%, N=727)	668/673 (99.26%, N=673)
TT-FF	486/487 (99.79%, N=487)	449/449 (100.00%, N=449)
TS-FL	1258/1290 (97.52%, N=1290)	1254/1282 (97.82%, N=1282)
TT-FL	139/149 (93.29%, N=149)	104/104 (100.00%, N=104)

Notes. Values are reported as correctly detected cases / manually evaluated samples (precision, total candidate pool size N).

7.3.3 Sensitivity of Parameters Related to Performance-saturated Conditions. To assess performance saturation, one straightforward criterion is whether the system reports a timeout error within a predefined threshold. A larger timeout (e.g., 300 seconds) is effective for filtering out extreme failures such as connection issues or unexpected hangs. However, it may miss practical saturation cases that occur earlier.

In our subjects, all normal URL requests are complete within 2 seconds. Therefore, a 300-second timeout is overly conservative and primarily captures abnormal failures rather than performance saturation. To improve sensitivity, we reduce the timeout threshold to 100 seconds. Although this lower threshold may introduce a small number of false positives, it enables the detection of additional performance-saturated conditions that would otherwise be missed.

Table 14 shows the sensitivity of the timeout parameter. When using a 300-second timeout, the precision is consistently high across all subjects, as only the most obvious saturation cases are detected. However, the number of detected conditions is limited. In contrast, the 100-second timeout slightly reduces precision but increases the number of detected cases (e.g., TS-FF: 727 vs. 673), demonstrating improved detection coverage. This highlights a trade-off between precision and sensitivity: a larger timeout yields more conservative but highly accurate detections, while a smaller timeout improves coverage at the cost of minor accuracy loss. Based on this observation, we adopt 100 seconds as a practical setting to balance detection coverage and accuracy.

In addition to timeout-based detection, we also consider a throughput-based criterion by comparing the observed throughput with the configured request frequency. We set the tolerance parameter β to 0, which represents the strictest condition for identifying saturation. This ensures that any deviation from the expected throughput is treated as a potential saturation signal. Nevertheless, in practice, a non-zero β can be used to delay detection and avoid premature alarms in scenarios with transient fluctuations.

Summary of Findings

SCPM effectively detects potential performance conditions (i.e., performance-equivalent, performance-spiked, and performance-saturated) with accuracy above 80% across most systems. Compared with the baseline methods, SCPM considers both exploration efficiency and search cost, enabling early identification of risky configuration–workload combinations without exhaustive testing. Performance-saturated conditions are detected most accurately because saturation thresholds are easier to identify in high workload frequency scenarios, while performance-spiked conditions are more sensitive to local prediction noise. Architectural complexity influences detection difficulty: systems with longer service traces exhibit more performance-equivalent and saturated cases, whereas simpler reference architectures show fewer cases, and the performance prediction is inevitably more challenging. Overall, integrating search-based methods and queueing models enables efficient and reliable prediction of diverse performance behaviors in microservice systems.

8 Threats to validity

8.1 Internal Validity

In our approach, since it is impossible to test all combinations of workloads and configurations, we evaluate our method in a subset of samples. Using different parts of the same dataset may yield different results. The large number of possible workload and configuration combinations creates uncertainty in the baseline and may affect the consistency of the comparison results. As a result, using a subset of the samples may affect our results. To ensure consistent results, we will conduct future work to explore performance features across workloads and configurations.

Moreover, our experiments are based on performance data collected from real-world testing. System noise may affect the accuracy of the results. Because processing time scales differ across services, noise at larger scales may mask the underlying data at smaller scales. We prevent this issue by assigning missions that have a similar level of noise to each service.

In addition, our method primarily uses QPN to analyze the relationship between software architecture and performance. However, as introduced in Section 2, the LQN is another possible model for estimating system performance. To this end, it is worth remarking that, while QPN requires information about interactions between the system's services, LQN requires more detailed knowledge, such as the definitions of processing resources and calls, which may increase the cost of building the model. Note that the selection of performance models and their variants remains an open and long-standing challenge in software performance engineering [2]. One of the primary concerns in practice is the prediction accuracy and applicability of the selected model for the target system. The presented experimental results support our decision to adopt QPN for evaluating the performance characteristics of configurable microservice systems.

8.2 Construct Validity

In our experiments, we record the processing time as an indicator of the system's performance, as this metric is considered one of the most complex in the field of performance modeling [27]. However, system performance is a broad concept that can also be affected by factors such as CPU utilization, memory consumption, and I/O behavior. Since our current measurement primarily focuses on processing time, other important performance aspects may not be fully captured. The three performance conditions consider only the processing time for combinations of workloads and configurations. This limitation may compromise the construct validity of our study, as our chosen metric reflects only one aspect of the overall system performance.

Furthermore, our current framework primarily focuses on time-related performance metrics, particularly processing time and response-time-related behaviors. Although the proposed hybrid modeling and condition-oriented exploration strategy can potentially be extended to other performance metrics, such as throughput, CPU utilization, memory consumption, or I/O-related behaviors, these metrics may require different modeling assumptions, workload representations, or condition definitions. For example, some performance metrics may exhibit different statistical characteristics or stronger dependencies on hardware and runtime environments, which could affect the applicability of the current QPN-based formulation. Extending the framework to support multiple heterogeneous performance metrics simultaneously remains an important direction for future work.

8.3 External Validity

In our research, we mainly use μBench to simulate the behavior of a microservice-based system. While μBench provides a controlled and flexible environment for evaluating performance models, it is still a simplified representation of real-world systems. Since we do not apply our approach to a large-scale microservice application, the results may not fully reflect the complexities and variability of real-world deployments, such as unpredictable network latency, workload diversity, and heterogeneous hardware. This limitation may reduce the external validity of our study, as the results might not generalize directly to real industrial systems.

In addition, the practicality of the proposed approach may depend on the characteristics of the target system. HQP is particularly suitable for configurable systems with relatively stable service boundaries, observable workload interactions, and identifiable architectural topologies. In such scenarios, workload and configuration interaction can be effectively modeled through QPN-based analysis. However, the approach may become less effective in highly dynamic environments where service topologies cannot be generated at runtime. In these cases, maintaining accurate queueing representations and service-level models may introduce additional modeling and monitoring overhead. Furthermore, deploying the approach in practice requires access to architecture information, which may not be readily available in industrial environments due to security issues.

9 Conclusion

In this paper, we present a configuration-workload-aware approach for performance modeling in large-scale configurable microservice systems. By leveraging Queueing Petri Nets (QPNs), our method captures the interactions among workloads, configurations, and system-level performance behaviors. Our empirical evaluation demonstrates that the proposed approach improves prediction accuracy while reducing testing costs compared with several state-of-the-art machine learning and deep learning baselines. Furthermore, the proposed condition-oriented exploration strategy efficiently identifies potential performance risks, including performance-equivalent, performance-saturated, and performance-spiked conditions, under dynamically changing workloads and configurations. We additionally discuss the application of the proposed method in real-world scenarios. SCPM is particularly suitable for configurable systems with workload-dependent service paths and large configuration spaces, where exhaustive testing is impractical. By integrating topology-aware QPN analysis with lightweight service-level prediction, the method can support efficient performance-risk exploration during benchmarking, deployment validation, or CI/CD processes. However, the approach currently relies on the availability of service topology and workload interaction information, and the accuracy of the QPN representation may depend on the validity of queueing assumptions under highly dynamic environments.

While the approach shows promising results in controlled experimental settings, future work will focus on evaluating the method on larger-scale real-world systems, incorporating additional

performance metrics beyond processing time, and investigating adaptive modeling strategies for highly dynamic architectures and workloads. Overall, our approach provides an effective and practical solution for improving the stability and reliability of performance analysis in configurable microservice systems.

Acknowledgment

This research was conducted by the SPEC RG DevOps Performance Working Group[9].

Data Availability

The code and dataset used in this study are available in the supplementary package[10].

References

[1] Liang Bao, Xin Liu, Ziheng Xu, and Baoyin Fang. 2018. AutoConfig: automatic configuration tuning for distributed message systems. In *ASE*. ACM, 29–40.
[2] André B Bondi. 2015. *Foundations of software and system performance engineering: process, performance modeling, requirements, testing, scalability, and practice*. Pearson Education.
[3] Tomás Cerný, Gabriel Goulis, and Amr S. Abdelfattah. 2025. Towards Change Impact Analysis in Microservices-based System Evolution. In *SANER*. IEEE, 159–169.
[4] Pengzhou Chen and Tao Chen. 2025. PromiseTune: Unveiling Causally Promising and Explainable Configuration Tuning. *CoRR* abs/2507.05995 (2025).
[5] Pengzhou Chen, Jingzhi Gong, and Tao Chen. 2025. Accuracy Can Lie: On the Impact of Surrogate Model in Configuration Tuning. *IEEE Trans. Software Eng.* 51, 2 (2025), 548–580.
[6] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. In *KDD*. ACM, 785–794.
[7] Jiezhu Cheng, Cuiyun Gao, and Zibin Zheng. 2023. HINNPerf: Hierarchical Interaction Neural Network for Performance Prediction of Configurable Systems. *ACM Trans. Softw. Eng. Methodol.* 32, 2 (2023), 46:1–46:30.
[8] Andrea Detti, Ludovico Funari, and Luca Petrucci. 2023. μBench: An Open-Source Factory of Benchmark Microservice Applications. *IEEE Trans. Parallel Distributed Syst.* 34, 3 (2023), 968–980.
[9] Diego Didona, Francesco Quaglia, Paolo Romano, and Ennio Torre. 2015. Enhancing Performance Prediction Robustness by Combining Analytical Modeling and Machine Learning. In *ICPE*. ACM, 145–156.
[10] Johannes Dorn, Stefan Mühlbauer, Stefan Jahns, Sven Apel, and Norbert Siegmund. 2026. Bayesian Multi-Level Performance Models for Multi-Factor Variability of Configurable Software Systems. In *Proceedings of the 48th IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE/ACM.
[11] Andy Field. 2013. *Discovering Statistics Using IBM SPSS Statistics* (4th ed.). SAGE Publications, London, UK.
[12] Greg Franks, Tariq Omari, C. Murray Woodside, Olivia Das, and Salem Derisavi. 2009. Enhanced Modeling and Solution of Layered Queueing Networks. *IEEE Trans. Software Eng.* 35, 2 (2009), 148–161.
[13] Jingzhi Gong and Tao Chen. 2023. Predicting Software Performance with Divide-and-Learn. In *ESEC/SIGSOFT FSE*. ACM, 858–870.
[14] Jingzhi Gong and Tao Chen. 2024. Predicting Configuration Performance in Multiple Environments with Sequential Meta-Learning. *Proc. ACM Softw. Eng.* 1, FSE (2024), 359–382.
[15] Jingzhi Gong, Tao Chen, and Rami Bahsoon. 2025. Dividable Configuration Performance Learning. *IEEE Trans. Software Eng.* 51, 1 (2025), 106–134.
[16] Jianmei Guo, Krzysztof Czarnecki, Sven Apel, Norbert Siegmund, and Andrzej Wasowski. 2013. Variability-aware performance prediction: A statistical learning approach. In *ASE*. IEEE, 301–311.
[17] Jianmei Guo, Jules White, Guangxin Wang, Jian Li, and Yinglin Wang. 2011. A genetic algorithm for optimized feature selection with resource constraints in software product lines. *J. Syst. Softw.* 84, 12 (2011), 2208–2221.
[18] Jianmei Guo, Dingyu Yang, Norbert Siegmund, Sven Apel, Atrisha Sarkar, Pavel Valov, Krzysztof Czarnecki, Andrzej Wasowski, and Huiqun Yu. 2018. Data-efficient performance learning for configurable systems. *Empir. Softw. Eng.* 23, 3 (2018), 1826–1867.
[19] Xiaofeng Guo, Xin Peng, Hanzhang Wang, Wanxue Li, Huai Jiang, Dan Ding, Tao Xie, and Liangfei Su. 2020. Graph-based trace analysis for microservice architecture understanding and problem diagnosis. In *ESEC/SIGSOFT FSE*. ACM, 1387–1397.

[9] SPEC RG DevOps Performance Working Group: https://research.spec.org/devopswg
[10] Supplementary package: https://zenodo.org/records/17584303

[20] Huong Ha and Hongyu Zhang. 2019. DeepPerf: performance prediction for configurable software with deep sparse neural network. In *ICSE*. IEEE / ACM, 1095–1106.

[21] Trevor Hastie, Robert Tibshirani, and Jerome H. Friedman. 2009. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction, 2nd Edition*. Springer.

[22] Christopher Henard, Mike Papadakis, Mark Harman, and Yves Le Traon. 2015. Combining Multi-Objective Search and Constraint Solving for Configuring Large Software Product Lines. In *ICSE (1)*. IEEE Computer Society, 517–528.

[23] Angelina Horn, Hamid Mohammadi Fard, and Felix Wolf. 2022. Multi-objective Hybrid Autoscaling of Microservices in Kubernetes Clusters. In *Euro-Par (Lecture Notes in Computer Science, Vol. 13440)*. Springer, 233–250.

[24] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. 2011. Sequential Model-Based Optimization for General Algorithm Configuration. In *LION (Lecture Notes in Computer Science)*. Springer, 507–523.

[25] Pooyan Jamshidi, Norbert Siegmund, Miguel Velez, Christian Kästner, Akshay Patel, and Yuvraj Agarwal. 2017. Transfer learning for performance modeling of configurable systems: an exploratory analysis. In *ASE*. IEEE Computer Society, 497–508.

[26] Christian Kaltenecker, Alexander Grebhahn, Norbert Siedmung, Jianmei Guo, and Sven Apel. 2020. Distance-Based Sampling of Software Configuration Spaces. In *SE (LNI, Vol. P-300)*. Gesellschaft für Informatik e.V., 61–64.

[27] Leonard Kleinrock. 1975. *Queueing Systems, Volume I: Theory*. Wiley-Interscience, New York.

[28] Samuel Kounev. 2006. Performance Modeling and Evaluation of Distributed Component-Based Systems Using Queueing Petri Nets. *IEEE Trans. Software Eng.* 32, 7 (2006), 486–502.

[29] Samuel Kounev and Alejandro P. Buchmann. 2006. SimQPN - A tool and methodology for analyzing queueing Petri net models by means of simulation. *Perform. Evaluation* 63, 4-5 (2006), 364–394.

[30] Rahul Krishna, Vivek Nair, Pooyan Jamshidi, and Tim Menzies. 2021. Whence to Learn? Transferring Knowledge in Configurable Systems Using BEETLE. *IEEE Trans. Software Eng.* 47, 12 (2021), 2956–2972.

[31] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. 2018. PerfFuzz: automatically generating pathological inputs. In *ISSTA*. ACM, 254–265.

[32] Luc Lesoil, Helge Spieker, Arnaud Gotlieb, Mathieu Acher, Paul Temple, Arnaud Blouin, and Jean-Marc Jézéquel. 2024. Learning input-aware performance models of configurable systems: An empirical evaluation. *J. Syst. Softw.* 208 (2024), 111883.

[33] Yufei Li, Liang Bao, Kaipeng Huang, and Chase Qishi Wu. 2025. CSAT: Configuration structure-aware tuning for highly configurable software systems. *J. Syst. Softw.* 222 (2025), 112316.

[34] Chieh-Jan Mike Liang, Zilin Fang, Yuqing Xie, Fan Yang, Zhao Lucis Li, Li Lyna Zhang, Mao Yang, and Lidong Zhou. 2023. On Modular Learning of Distributed Systems for Predicting End-to-End Latency. In *NSDI*. USENIX Association, 1081–1095.

[35] Lizhi Liao. 2023. Addressing Performance Regressions in DevOps: Can We Escape from System Performance Testing?. In *45th IEEE/ACM International Conference on Software Engineering: ICSE 2023 Companion Proceedings, Melbourne, Australia, May 14-20, 2023*. IEEE, 203–207.

[36] Lizhi Liao, Jinfu Chen, Heng Li, Yi Zeng, Weiyi Shang, Jianmei Guo, Catalin Sporea, Andrei Toma, and Sarah Sajedi. 2020. Using black-box performance models to detect performance regressions under varying workloads: an empirical study. *Empir. Softw. Eng.* 25, 5 (2020), 4130–4160.

[37] Lizhi Liao, Simon Eismann, Heng Li, Cor-Paul Bezemer, Diego Elias Costa, André van Hoorn, and Weiyi Shang. 2025. Early Detection of Performance Regressions by Bridging Local Performance Data and Architectural Models. In *47th IEEE/ACM International Conference on Software Engineering, ICSE 2025, Ottawa, ON, Canada, April 26 - May 6, 2025*. IEEE, 2841–2853.

[38] Lizhi Liao, Heng Li, Weiyi Shang, Catalin Sporea, Andrei Toma, and Sarah Sajedi. 2023. Adapting Performance Analytic Techniques in a Real-World Database-Centric System: An Industrial Experience Report. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*. ACM, 1855–1866.

[39] Max Lillack, Johannes Müller, and Ulrich W. Eisenecker. 2013. Improved Prediction of Non-functional Properties in Software Product Lines with Domain Context. In *Software Engineering (LNI, Vol. P-213)*. GI, 185–198.

[40] Lin Ma, Dana Van Aken, Ahmed Hefny, Gustavo Mezerhane, Andrew Pavlo, and Geoffrey J. Gordon. 2018. Query-based Workload Forecasting for Self-Driving Database Management Systems. In *SIGMOD Conference*. ACM, 631–645.

[41] Hugo Martin, Mathieu Acher, Juliana Alves Pereira, Luc Lesoil, Jean-Marc Jézéquel, and Djamel Eddine Khelladi. 2022. Transfer Learning Across Variants and Versions: The Case of Linux Kernel Size. *IEEE Trans. Software Eng.* 48, 11 (2022), 4274–4290.

[42] Microsoft Ignite. 2025. *Microservices architecture style*. ttps://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices Accessed: 2025-11-17.

[43] Stefan Mühlbauer, Florian Sattler, Christian Kaltenecker, Johannes Dorn, Sven Apel, and Norbert Siegmund. 2023. Analysing the Impact of Workloads on Modeling the Performance of Configurable Software Systems. In *ICSE*. 2085–2097.

[44] Vivek Nair, Tim Menzies, Norbert Siegmund, and Sven Apel. 2017. Using bad learners to find good configurations. In *ESEC/SIGSOFT FSE*. ACM, 257–267.

[45] Vivek Nair, Zhe Yu, Tim Menzies, Norbert Siegmund, and Sven Apel. 2020. Finding Faster Configurations Using FLASH. *IEEE Trans. Software Eng.* 46, 7 (2020), 794–811.

[46] Jeho Oh, Don S. Batory, Margaret Myers, and Norbert Siegmund. 2017. Finding near-optimal configurations in product lines by random sampling. In *ESEC/SIGSOFT FSE*. ACM, 61–71.

[47] Riccardo Pinciroli, Aldeida Aleti, and Catia Trubiani. 2023. Performance Modeling and Analysis of Design Patterns for Microservice Systems. In *ICSA*. IEEE, 35–46.

[48] Nyyti Saarimäki, Mikel Robredo, Valentina Lenarduzzi, Sira Vegas, Natalia Juristo, and Davide Taibi. 2025. Does microservice adoption impact the velocity? A cohort study. *Empir. Softw. Eng.* 30, 5 (2025), 130. https://doi.org/10.1007/S10664-025-10673-7

[49] Safdar Aqeel Safdar, Hong Lu, Tao Yue, and Shaukat Ali. 2017. Mining cross product line rules with multi-objective search and machine learning. In *GECCO*. ACM, 1319–1326.

[50] Mohammad Reza Saleh Sedghpour, Aleksandra Obeso Duque, Xuejun Cai, Björn Skubic, Erik Elmroth, Cristian Klein, and Johan Tordsson. 2023. HydraGen: A Microservice Benchmark Generator. In *CLOUD*. IEEE, 189–200.

[51] Mohammad Shahrad, Rodrigo Fonseca, Iñigo Goiri, Gohar Irfan Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *USENIX ATC*. USENIX Association, 205–218.

[52] Weiyi Shang, Ahmed E. Hassan, Mohamed N. Nasser, and Parminder Flora. 2015. Automated Detection of Performance Regressions Using Regression Models on Clustered Performance Counters. In *ICPE*. ACM, 15–26.

[53] Yasir Shoaib and Olivia Das. 2011. Web Application Performance Modeling Using Layered Queueing Networks. In *PASM@ICPE (Electronic Notes in Theoretical Computer Science, Vol. 275)*. Elsevier, 123–142.

[54] Yangyang Shu, Yulei Sui, Hongyu Zhang, and Guandong Xu. 2020. Perf-AL: Performance Prediction for Configurable Software through Adversarial Learning. In *ESEM*. ACM, 16:1–16:11.

[55] Norbert Siegmund, Alexander Grebhahn, Sven Apel, and Christian Kästner. 2015. Performance-influence models for highly configurable systems. In *ESEC/SIGSOFT FSE*. ACM, 284–294.

[56] Daniel Sokolowski, Pascal Weisenburger, and Guido Salvaneschi. 2021. Automating serverless deployments for DevOps organizations. In *ESEC/SIGSOFT FSE*. ACM, 57–69.

[57] Jingwei Sun, Guangzhong Sun, Shiyan Zhan, Jiepeng Zhang, and Yong Chen. 2020. Automated Performance Modeling of HPC Applications Using Machine Learning. *IEEE Trans. Computers* 69, 5 (2020), 749–763.

[58] Catia Trubiani, Riccardo Pinciroli, Andrea Biaggi, and Francesca Arcelli Fontana. 2023. Automated Detection of Software Performance Antipatterns in Java-Based Applications. *IEEE Trans. Software Eng.* 49, 4 (2023), 2873–2891.

[59] Bhuvan Urgaonkar, Giovanni Pacifici, Prashant Shenoy, Mike Spreitzer, and Asser Tantawi. 2005. An analytical model for multi-tier internet services and its applications. *ACM SIGMETRICS Performance Evaluation Review* 33, 1 (2005), 291–302.

[60] Miguel Velez, Pooyan Jamshidi, Norbert Siegmund, Sven Apel, and Christian Kästner. 2021. White-Box Analysis over Machine Learning: Modeling Performance of Configurable Systems. In *ICSE*. IEEE, 1072–1084.

[61] Jóakim von Kistowski, Simon Eismann, Norbert Schmitt, André Bauer, Johannes Grohmann, and Samuel Kounev. 2018. TeaStore: A Micro-Service Reference Application for Benchmarking, Modeling and Resource Management Research. In *MASCOTS*. IEEE Computer Society, 223–236.

[62] Emilien Wansart, Maxime Goffart, Justin Iurman, and Benoit Donnet. 2024. MSTG: A Flexible and Scalable Microservices Infrastructure Generator. In *TMA*. IEEE, 1–4.

[63] C. Murray Woodside, Greg Franks, and Dorina C. Petriu. 2007. The Future of Software Performance Engineering. In *International Conference on Software Engineering, ISCE 2007, Workshop on the Future of Software Engineering, FOSE 2007, May 23-25, 2007, Minneapolis, MN, USA*, Lionel C. Briand and Alexander L. Wolf (Eds.). IEEE Computer Society, 171–187. https://doi.org/10.1109/FOSE.2007.32

[64] Yuanjie Xia, Zishuo Ding, and Weiyi Shang. 2023. CoMSA: A Modeling-Driven Sampling Approach for Configuration Performance Testing. In *ASE*. IEEE, 1352–1363.

[65] Yuanjie Xia, Lizhi Liao, Jinfu Chen, Heng Li, and Weiyi Shang. 2024. Reducing the Length of Field-Replay Based Load Testing. *IEEE Trans. Software Eng.* 50, 8 (2024), 1967–1983.

[66] Yi Xiang, Han Huang, Yuren Zhou, Sizhe Li, Chuan Luo, Qingwei Lin, Miqing Li, and Xiaowei Yang. 2022. Search-based Diverse Sampling from Real-world Software Product Lines. In *ICSE*. ACM, 1945–1957.

[67] Zezhen Xiang, Jingzhi Gong, and Tao Chen. 2025. Dually Hierarchical Drift Adaptation for Online Configuration Performance Learning. *CoRR* abs/2507.08730 (2025).

[68] Chenxi Zhang, Xin Peng, Chaofeng Sha, Ke Zhang, Zhenqing Fu, Xiya Wu, Qingwei Lin, and Dongmei Zhang. 2022. DeepTraLog: Trace-Log Combined Microservice Anomaly Detection through Graph-based Deep Learning. In *ICSE*. ACM, 623–634.

[69] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Wenhai Li, and Dan Ding. 2021. Fault Analysis and Debugging of Microservice Systems: Industrial Survey, Benchmark System, and Empirical Study. *IEEE Trans. Software Eng.* 47, 2 (2021), 243–260.