Learning-based Automated Generation of Critical Workload Configurations for Microservices Performance Testing

***Abstract*—Performance testing is an essential activity in the engineering of microservice applications to identify deviations from the specified ranges of relevant metrics and to analyse resources usage. It demands for high automation to fit within the short microservices development-operation cycles. Engineers are often interested in identifying critical workloads - ideally, in the "minimal" load configurations causing tests to expose performance issues. Triggering performance issues is challenging, requiring proper workload characterization and test design.**

We present a framework for learning-based automated generation of critical performance testing workloads for microservices. The framework can harness various learning strategies: we analyze a Deep Neural Network, a Large Language Model and a Causal Reasoning strategy. We evaluate them experimentally on four subjects, using a random approach and a manually-crafted ground truth as baselines. The results show that the strategies exhibit different behavior depending on the data they learn from. When inferring from past executions data including performance issues, the causal model performs better. The random predictor is preferable when no data is available; however, it is more costly as it requires more tests. The results allow to draw practical recommendations for testers on how to select the most suitable strategy depending on the needs.

***Index Terms*—Microservices, Performance testing, Machine Learning, Large Language Models, Causal reasoning**

I. Introduction

Performance testing of microservice architectures is essential to understand how business workloads impact performance and resources usage. It focuses on evaluating performance aspects under varying load conditions and diverse deployment methods [1], and is valuable, as for all applications running in today data centers, for capacity planning (making informed decisions about scaling infrastructure), for prevention (defining proactive measures preventing performance issues), and for analyzing resources usage [2].

Besides basic *load testing* – to verify satisfaction of performance requirements under expected usage loads (*operational workload*) - an important goal of microservices performance testing is to expose performance failures. Usually, test engineers identify parameters characterizing workload configurations (e.g., types of requests, intensity, usage scenarios), this way defining a *workload model*; then they run tests by varying such parameters, namely looking for failure-exposing workload configurations [3]. Ideally, engineers are interested in finding the "minimal" configurations that cause performance failures, namely configurations in which the system starts experiencing a degradation, even with a relatively low load. These critical configurations are sometimes called "knee points" of the system load-performance curve, and are important for performance engineers to understand what is the minimum load such that the system performs without degradation – that is useful, for instance, for capacity planning. Finding such critical configurations is a costly process, requiring multiple test sessions to explore the space of workload configurations. A naïf approach would be to test stressful conditions incrementally, starting from a nominal or a minimal workload configuration. But such an approach can be extremely expensive, depending on how large the workload configurations space is.

Due to their characteristics and the short release cycles, especially in DevOps contexts [4], microservices pose specific challenges in performance testing with respect to other distributed software architectures [5]. Automated mechanisms for detecting performance anomalies are instrumental in optimizing testing costs [1]. While tools exist today (e.g., `Locust`),[1] that ease basic tasks like running tests given a manual specification, solutions for automated performance tests generation or selection still lack. Researchers have mainly focused on assessing the performance of deployment alternatives [6], [7], on performance forecasting [8], on performance degradation prediction [9], and on the design of performance testing platforms [10]. This has been done with the goal of (semi-)automation of tasks like: the execution of tests with manually generated workloads; the generation of workloads from manually specified configurations, or the execution of load tests as part of continuous integration/delivery pipelines.

We present `microWave`, a methodological framework harnessing Artificial Intelligence (AI) strategies to find workloads that expose performance failures in microservices performance testing. It supports testers with the automatic generation of workloads that make the system perform improperly, particularly searching for minimal critical workloads.

We compare three strategies based on: Machine Learning (ML), in the form of a Deep Neural Network (`DNN`) and a Large Language Model (`LLM`), and a Causal Model (`CM`). We evaluate them experimentally, comparing them against each other as well as against a random predictor and a manual technique. The experiments use four subjects, to analyze the strategies on different kinds of microservice applications.

The experimental results show that when models can learn

[1] `Locust` is an open source performance/load testing tool, meant to ease running load tests distributed over multiple machines (https://locust.io).

from data of past executions which include performance issues, the causal reasoning strategy performs better than the others. The random predictor is preferable when prior data about performance issues is not available; however, it is more costly as it requires more tests. We also draw from results recommendations for practitioners on how to select the most suitable strategy depending on their needs.

In summary, the work provides the following contributions:

- A framework for the automated generation of workload configurations for microservice performance testing, that harnesses various AI learning strategies;
- Experimental comparison of three learning strategies representative of different categories of techniques for the stated goal (first such study, to the best of our knowledge);
- Practical guidelines for testers to select the learning strategy best suiting their specific context;
- Framework and study artifacts are made publicly available, for repeatability and further experiments.

The paper is organized as follows. Section II discusses related work. Section III describes the proposed framework. Section IV sets the research questions and the evaluation metrics. Section V describes experiments and results. Section VII draws practical guidelines from results. Section VIII illustrates threats to validity. Section IX contains concluding remarks.

II. Related work

Microservices have been argued to demand for *ad hoc* performance testing approaches with respect to other architectural paradigms, including Service Oriented Architectures, of which they are considered an advancement [5]. Despite this, microservices performance testing is a relatively little investigated research area. Vasilevskii *et al.* [10] emphasize the problems faced in companies by dedicated performance teams, who often manually craft in-house solutions, and they advocate the need for shared practices and common tools.

Most existing literature exploits past data to support microservice tests generation/execution mimicking the observed behaviour. The common idea is to feed a model with what has been observed in operation and use it to generate tests.

Cooper *et al.* [11] propose a framework leveraging realistic usage traces to identify and execute a concise set of performance tests. The goal is to replicate the response time distribution with shorter test sessions: the authors replicated the response time distribution of a 24-hour test within a five-minute test, achieving a negligible error percentage.

Camilli *et al.* [12] employ probabilistic model checking to identify performance issues within microservices and correlate them to system-level requirements. This approach builds a formal model, a Continuous Time Markov Chain, through a Bayesian inference process that assimilates data from multiple load testing sessions, aiming to replicate realistic workload conditions. The resulting formal model is leveraged to automate the verification using probabilistic model checking techniques. Additionally, it enables the calculation of a configuration score, facilitating the evaluation of diverse deployment alternatives.

Camilli *et al.* [13] addressed joint performance and reliability *ex vivo* testing , proposing a methodology and support platform (MIPaRT) that fit into DevOps processes. MIPaRT leverages usage and system data from past operational (Ops) phases to partially automate tests at a DevOps decision gate; it then visualizes performance indicators, helping to pinpoint performance and reliability issues in an integrated way.

De Camargo *et al.* [14] aim at automating the execution of manually specified tests. They propose an architecture that incorporates test specifications into each microservice, thereby enabling the automated execution of performance tests. It provides a structured approach for defining test specifications, encapsulating the information necessary to compose test requests, and a mechanism to attach and expose these specifications. The proposed architecture does not impact service performance, enabling its deployment in production environments for realistic performance evaluation.

Avritzer *et al.* aim at evaluating the performance of deployment alternatives with load tests [6], to assess their scalability [7]. Load tests are based on the *operational profile*, derived in a preliminary step from past data. A specific metric quantifies a configuration's ability to meet scalability requirements under that profile. Chosen a workload from the profile and a configuration, tests consist of runs to first compute reference values of performance metrics (like average response time), to then assess scalability under increasing loads.

Machine Learning models in microservices performance testing, aiming for *performance prediction*, have been investigated by Santos *et al.* [8], who analyzed six models (ARIMA, Multilayer Perceptron, Support Vector Regression, Random Forest, Long Short-Term Memory, eXtreme Gradient Boosting), and by Grohmann *et al.* [9], who proposed SuanMing, a framework for predicting root causes for anticipated performance degradations in cloud settings. Both these forecasting techniques do not generate test workloads. ML models used for predictions are trained from available workload data, e.g., tracing data [8], or from usage time series synthetically generated or collected by monitoring past executions [9]. In a preliminary study, Giamattei *et al.* [15] explore the use of causal models to determine the number of users needed to trigger performance issues in a microservices architecture under predefined workload configurations Finally, the advent of LLMs opens "a new era in microservices testing, blending automation with intelligent capabilities" [1]; however, the potential of LLMs to address challenges in microservices testing remains to be explored.

We propose to use AI models in microservices performance testing for automatic generation of workload configurations likely to expose performance issues. To this aim we present a methodological framework (`microWave`) and we analyze three types of learning strategies: a `DNN`, a `LLM` and a Causal Reasoning technique using a `CM`.

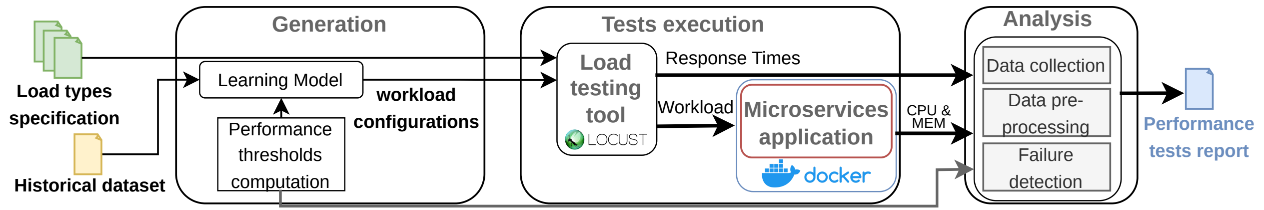


Fig. 1: The automated testing workflow of the `microWave` framework

III. The microWave framework

We consider a *performance issue* to occur any time one or more observed performance metrics exceed defined *thresholds*. Thresholds may be provided by the tester, or automatically computed by the framework on a dataset of past tests or past executions like in Avritzer *et al.* [7]. `microWave`: *i)* finds a minimal critical workload configuration (defined later); *ii)* generates a workload according to the configuration found; *iii)* parses the test results to determine if any performance issue actually occurred.

Figure 1 illustrates the proposed framework, which comprises three phases: *i) Generation*, enabled by the *learning models*; *ii) Tests Execution*, where the generated configuration is utilized by a *load testing tool* to run the workload on the microservices application; and *iii) Analysis*, consisting in *data collection* and *pre-processing*, and *failure detection*.

`microWave` requires the definition of an operational profile (in a `locustfile`), containing a characterization of how the user interacts with the system, acting as a specification for the workload to generate. It consists of:

- a list of *user types* (named `HttpUser` in `Locust`); each user type issues (a sequence of) requests to accomplish a task (e.g., purchase of a product);
- for each *user type*, its occurrence probability (i.e., the probability that the system will be solicited with the sequence of requests of that user type, referred to as *user spawn probability*).

An excerpt of the profile is shown in Listing 1.

Engineers can define various types of profiles, called *load types* in the following, e.g., a *uniform* profile, where every user type has the same probability of occurrence, or a profile *unbalanced* toward a specific *user type*. Like for performance thresholds, the operational profile may be defined by the tester, or extracted from data gathered from monitoring test executions – which are readily available in a DevOps context - by clustering the sequences of requests. In this case, the probability would be estimated as the frequency of occurrence of sequences.

The framework leverages the historical dataset to parameterize the learning models. The historical dataset used in our study comprises the following (*controllable*) parameters that define the workload:

- *User size* (US), the number of users concurrently sending requests to the system;

```
class UserNoLogin(HttpUser):
    weight = 70  #User spawn probability
    @task
    def perform_task(self):
        #Request sequence
        operations=["home", "getCatalogue",
        ↪ "getItem", "getRelated"]
        #Request sequence run
        perform(self, operations)
class UserLoginAndShop(HttpUser):
    weight = 15
    @task
    def perform_task(self):
        operations=["home", "login", "getCatalogue",
        ↪ "getItem", "addToCart", "createOrder",
        ↪ "viewOrdersPage", "getOrders"]
        perform(self, operations)
class UserLoginAndCheckCart(HttpUser):
    weight = 15
    @task
    def perform_task(self):
        operations=["home", "login", "home",
        ↪ "getCatalogue", "getItem", "addToCart",
        ↪ "home", "getCart"]
        perform(self, operations)
```

Listing 1: Excerpt from a sample `locustfile`

- *Load type* (LT), defining the different ways (user profiles) through which users can interact with services, defined categorically (e.g., unbalanced towards *checkout*, unbalanced towards checkout *cart*, uniform, etc.);
- *Spawn rate* (SR), the rate to spawn users (number of users per second);

and the following (*observable*) performance metrics for each microservice:

- Resource usage metrics, namely *CPU usage* (CPU) and *used memory* (MEM);
- *Response time* (RT), the time elapsed between sending the request and receiving the response;
- *Request rate* (RR), the number of requests processed per second;

Given a workload specification as in Listing 1 above, `microWave` searches the space of **workload configurations**, defined as triples <*US, LT, SR*>. A configuration causing a performance issue is said *critical*. A **minimal critical configuration** is a *critical configuration* such that there exists no other critical configuration with lower *US* and smaller *SR*. There can be multiple minimal critical configurations. `microWave` finds one such configuration by querying a model (we here

experiment DNN, LLM, CM), to explore the space of possible configurations without actually executing them. In presence of multiple minimal critical configurations, we opt for the one with smaller *US*, since the user size has been observed to have greater impact than the spawn rate.

The generated configuration is then passed to the locust tool, to actually run the microservice application with the specified workload.

Let us now describe in detail each phase within the microWave framework.

Generation. This phase focuses on automatically generating a minimal critical configuration using a learning model. The experimented models are trained using historical data, except for the LLM, which is pre-trained.

After training, the DNN is used for predicting the expected response Y when the input **X** (namely, $X_1 = US$, $X_2 = LT$ and $X_3 = SR$) happens to be equal to a given value **x**, i.e.: $P(Y|X_1 = x_1, X_2 = x_2, X_3 = x_3)$.

The CM is automatically built with a Causal Structure Discovery (CSD) algorithm [16], as in Giamattei *et al.* [15]. CSD algorithms extract the causal relationships between variables, usually in the form of a Directed Acyclic Graph (DAG), where nodes are the variables and edges represent cause-effect relationships among them. Furthermore, they can optionally be aided by prior knowledge about the specific domain. For instance, in a microservices architecture, prior knowledge can be automatically derived from its service mesh - that describes the services' dependencies - to predetermine relations (e.g., forbidden or required edges). In our study, we have exploited the following relations:

- $US, LT, SR \rightarrow RR_s$, where US, LT, SR are controllable variables and RR_s is the request rate of every service s;
- $RR_s \rightarrow CPU_s, MEM_s, RT_s$ for every service s;
- $CPU_s, MEM_s, RT_s, RR_s \rightarrow CPU_{s'}, MEM_{s'}, RT_{s'}, RR_{s'}$ for every dependence $s \rightarrow s'$ in the service mesh.

One of the most commonly used types of CM - employed also in this study - is the Structural Causal Model, a DAG where the relationships between variables $x_i \in X$ are described as a collection of structural assignments $x_i := f_i(Pa(x_i), u_i)$ that define the (endogenous) random variables x_i as a function of their parents $Pa(x_i)$ and of (exogenous) independent random noise variables u_i. The CM is queried to predict the expected response Y when the input X is actively set to a given value (i.e.: $P(Y|do(X_1 = x_1, X_2 = x_2, X_3 = x_3))$, according to the *do* operator [17], yielding a free-from-confounders prediction.

The LLM is employed as a domain expert, thereby exploiting the information acquired during the pre-training phase to identify the critical configurations. The LLM model utilizes historical data through Retrieval-Augmented Generation (RAG). RAG is a method commonly used to adapt a general-purpose model, like an LLM, to specific use cases [18]. This involves augmenting the LLM prompt with a specific "context", in our case derived from observational data about the system behavior. The LLM is queried via few-shot prompting (example in Figure 2): the prompt is constructed incorporating information extracted from the dataset using RAG, a set of constraints, K positive/negative examples (K-shot prompt), and the question. The constraints specify the boundaries for the values of the configuration parameters. The prompt contains K examples, each defining a performance metric and its threshold. Positive examples include an affirmative answer ("YES") and the configuration causing the performance issue; negative examples consist solely of a negative response ("NO"). Finally, the LLM is asked to complete a partial example given in input (e.g., RT of s_1 exceeds the threshold) by trying to identify a minimal critical configuration. The distribution of the positive/negative examples follows the distribution of the performance issues in the dataset (e.g., for $K = 5$ and if 40% of the configurations in the historical dataset lead to performance issues, we provide the model with 2 positive and 3 negative examples).

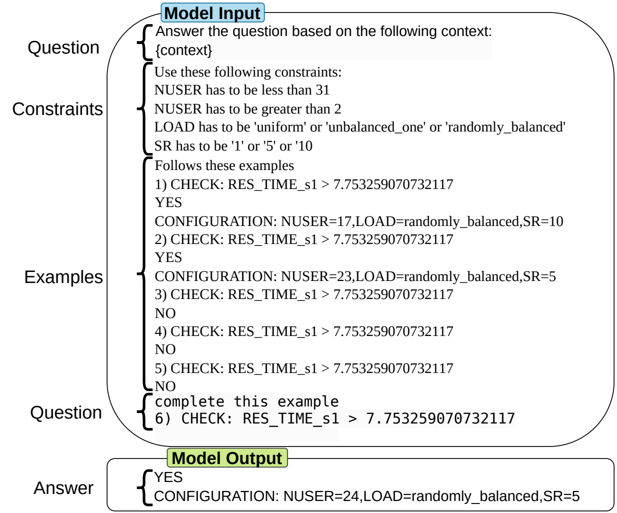


Fig. 2: Example LLM prompt

DNN and CM are used to explore the workload configuration space avoiding test executions. They replace the test execution by predicting its outcomes. Specifically, inference (for DNN) and intervention (for CM) are employed to predict the performance metrics under a workload configuration. To explore the configuration space, US is gradually increased, while LT and SR change. The exploration terminates once a critical configuration is identified, i.e., when the model's prediction exceeds a predefined threshold for at least one performance indicator of the specified service(s).

To enable such an exploration, we restrict US to a range $[US_{min}, US_{max}]$ (US={US_{min}, US_{min+1}, US_{min+2}, ..., US_{max}}), while treating both LT and SR as categorical variables to avoid state space explosion (e.g., LT={*Uniform, Unbalanced*}, SR={SR_{Low}=1, SR_{Med}=5, SR_{High}= 10}).

Tests Execution. The AI generated configurations are used by locust to actually produce the test workload. Then, the application is run with the workload, and, during the execution, *RR*, *RT*, *CPU*, and *MEM* statistics are collected. The first two metrics are gathered by locust, the others are collected through the Dockerstats tool.

Analysis. After the workload execution is completed, the performance evaluator analyzes the collected results. This

component examines the executed tests (i.e., workload configurations) to identify any exposed performance issues. A performance issue is detected whenever (at least) one observable performance variable's value exceeds a predefined threshold. Thresholds can be determined using historical data or by running an ideal workload with a single user to monitor performance variables (e.g., RT, CPU, and MEM). Thresholds are defined according to Avritzer *et al.* [7] (*scalability thresholds*) as $\tau_X = \mu_X + 3 \cdot \sigma_X$, where X is the variable of interest, and μ_X and σ_X are its mean and standard deviation over past executions.

IV. Research questions and metrics

This study addresses the following research questions:

RQ1: What are the effectiveness and the efficiency of learning models when trained with data including samples of performance issues?

RQ2: What are the effectiveness and the efficiency of learning models when trained with data not including samples of performance issues?

RQ3: What is the time efficiency of different learning models?

For evaluating surrogate models effectiveness, we rely on the metrics *precision* and *recall*, commonly used for evaluating classification models. The framework acts as a binary classifier, establishing whether or not a given configuration yields a performance issue. *Precision* is the ratio of performance issues correctly detected (True Positives) to the number of all potential performance issues predicted by the model (True Positives + False Positives). *Recall* is the ratio of performance issues correctly detected to the number of actual performance issues (True Positives + False Negatives).

As for efficiency, we compute the distance between the predicted configurations and the minimal configuration. As ground truth, we preliminarily run all the $|US| \times |LT| \times |SR|$ configurations, and took the minimal configuration as defined in Section III. The distance D sums up the distances in terms of US (D_0), LT (D_1), and SR (D_2), as follows:

$$D = D_0 + D_1 + D_2 \tag{1}$$

$$D_0 = \frac{1}{3} \frac{|N_p - N_r|}{(U_{max} - U_{min})} \tag{2}$$

$$D_1 = \begin{cases} \frac{1}{3}, & \text{if } LT_p \neq LT_r, \\ 0, & \text{otherwise} \end{cases} \tag{3}$$

$$D_2 = \begin{cases} \frac{1}{3}, & \text{if } SR_p \neq SR_r, \\ 0, & \text{otherwise} \end{cases} \tag{4}$$

where $N \in (U_{min}, U_{max})$ is the *US*; p and r are the predicted and the minimal configuration, respectively. We compute the distance in two cases: *i)* when the predicted configuration corresponds to an actual performance issue (True Positive), and *ii)* when it does not cause a performance issue (False Positive). The former is calculated considering only the configurations generated that result in performance issues; the latter is calculated considering only the configurations that do not result in performance issues.

To evaluate the time efficiency of the learning models, we measure the time required for model parametrization and for configuration generation. Specifically, for `DNN`, this includes the training time of the neural network; for `LLM`, the context creation time; for `CM`, the causal model construction time. Additionally, the inference time is measured for all models.

V. Experiments design

A. Subjects

The experimentation uses four microservices applications, with the goal of evaluating the performance of the learning strategies on different kinds of subjects.

`µBench`[2] is a microservice application built via the homonymous tool, which creates customizable applications. The application considered is composed of 10 services, with a random service mesh and a random stressing function for each service (stress/idle function).

`SockShop`[3] is an application for demonstration and testing of microservices and cloud-native technologies. It is composed of 8 services. Its services are loosely interconnected (differently from other subjects), implying that each service handles an entire functionality, resulting in a service mesh with few explicit dependencies. This complicates the construction of the `CM`, as prior knowledge contains few mandatory edges.

`Online Boutique`[4] is an application composed of 12 services, emulating a web-based e-commerce app. To emulate a background load, it includes a load generator service, which we exclude in our experiments to control request volume.

`TeaStore`[5] is a reference application for microservices benchmarks and tests. It simulates a basic web store, automatically generating customer orders. It comprises a registry and 5 services, communicating via REST API.

B. Baselines

`microWave` is experimented in three variants, corresponding to the usage of the three types of learning models (`DNN`, `LLM`, `CM`). These are compared against a random predictor and against a manual technique.

Random Predictor (RP): For generating configurations, `RP` selects randomly a *US*, a *LT* and a *SR*, and, for every service, generates three configurations (one each for RT, CPU, MEM).

Manual ground truth: Performance testing is traditionally performed by submitting several workloads to the system, which is monitored to gather performance indicators. A strategy to spot critical configurations is to test all the possible workload conditions. In our case, the workload is defined by three factors: US, LT and SR. Considering the last two as categorical, a tester has to manually increment US and then change the other two factors until a performance issue is detected (fail-stop strategy). Incrementing US by 1 ensures to find all the critical configurations, hence setting a ground truth (*Precision* $=$ *Recall* $= 1$), but it is extremely expensive.

[2]https://github.com/mSvcBench/muBench.

[3]https://github.com/ocp-power-demos/sock-shop-demo.

[4]https://github.com/GoogleCloudPlatform/microservices-demo.

[5]https://github.com/DescartesResearch/TeaStore.

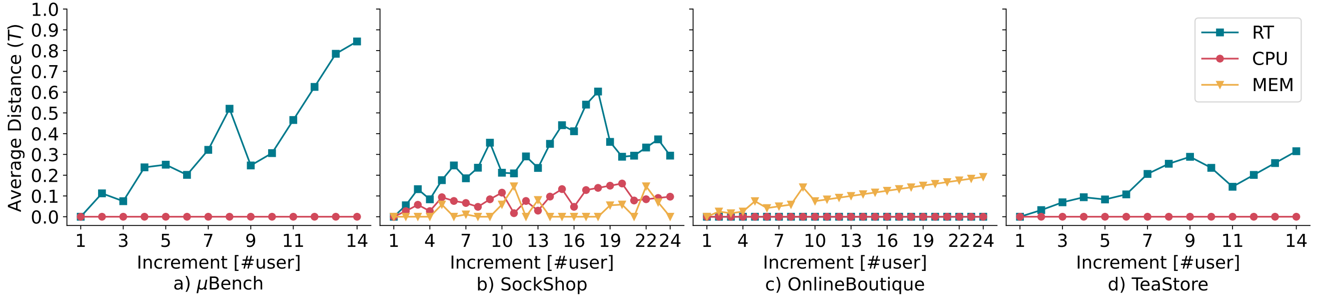


Fig. 3: Average distance between detected and actual critical configurations for various increments of the number of users

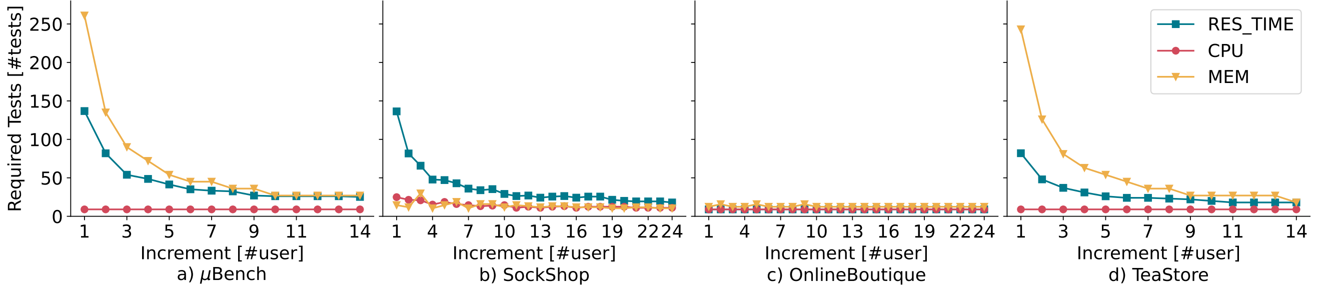


Fig. 4: Number of tests required to detect a critical configuration for various increments of the number of users

We implement it to measure the potential effectiveness/efficiency trade-off of `microWave` with respect to the ground truth. In particular, we can assess the loss of precision and recall obtained by `microWave` compared to this ground truth (i.e., maximum effectiveness) *vs.* the gain in terms of number of test cases. In fact, while with `microWave` we run only one test case suggested by the model (obtaining a *Precision/Recall*$\leq$ 1), the manual strategy needs many test cases to explore all the combinations of the three factors ($|US| \times |LT| \times |SR|$) test cases per service-performance metric pair), representing the cost for reaching the maximum effectiveness.

Given the high cost, we also considered variants of this manual strategy with an increment step for US greater than 1. Figure 3 and Figure 4 show how the distances of the configurations detected with respect to the minimal configuration and the number of tests required to detect a critical configuration change with the increment step. When the increment step is 1 the distance equals zero (Figure 3), but requires a large number of tests (Figure 4). As we increase the step size, the number of tests required to detect a critical configuration reduces, but also reduces the minimality of the configurations detected (i.e., the distance increases). In addition, with an increment step greater than 1, some performance issues can be missed. This can lead to a scenario where a resource's performance peaks under a specific workload, but that peak goes undetected because the condition which caused it is not tested. For this reason, we consider the variant with increment step equal to 1 as ground truth, for which we have *Precision* = *Recall*= 1, distance $T = 0$, and number of required tests, depending on the subject, equal to $|US| \times |LT| \times |SR|$ per service-performance metric pair (first point of Figure 4).

C. Experimental methodology

A historical dataset is created for each subject. Data is gathered running various workloads, each defined by a configuration: $< US, LT, SR >$. US is uniformly distributed. We use three Load Types, and we consider three values for the Spawn Rate {1,5,10}. The LT is one-hot encoded. As the applications handle requests of different nature, specific load profiles are defined for each of them.

For `µBench` the three LT are defined as in Ref. [15]:

- *uniform*: users send requests to services sequentially;
- *unbalanced*: one service has a higher probability of receiving a request (load unbalanced towards one service);
- *randomly balanced*: the invocation matrix is randomly generated, the load is balanced; any service $s \in S$ has the same probability $\frac{1}{|S|}$ of being invoked.

The other three subjects emulate the core functionalities of an e-commerce application, namely cart management (adding and removing items) and checkout. The three LT are:

- *normal*: the nominal workload received;
- *stress cart*: stress workload in a cart management scenario, with many users concurrently adding/removing products from their carts;
- *stress checkout*: stress workload in a checkout scenario, with many users concurrently completing purchases.

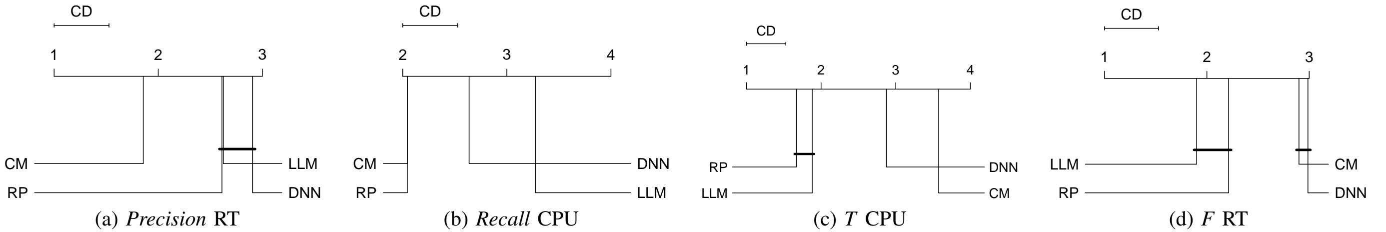
Each configuration is run five times and lasts three minutes. There is a two-minute pause between runs of two configurations, to avoid carryover effects (consecutive runs influencing each other, e.g., in CPU and MEM).

As for the three learning models, the `DNN` is a neural network consisting of two hidden layers, each containing 64 units, employing the ReLU activation function. As `LLM` model we consider `phi-3-mini` [19] queried with a 5-shot

TABLE I: RQ1: Models comparison (datasets with samples of past performance issues)

Performance metric	Model	µBench				SockShop				Online Boutique				TeaStore			
		P	*R*	*T*	*F*	*P*	*R*	*T*	*F*	*P*	*R*	*T*	*F*	*P*	*R*	*T*	*F*
RT	DNN	0.34	0.95	**0.07**	0.32	0.56	1.00	**0.29**	0.32	0.99	**1.00**	**0.00**	0.33	0.02	0.15	**0.00**	**0.11**
	LLM	0.42	**1.00**	0.28	0.42	0.50	1.00	0.51	0.49	**1.00**	0.97	0.18	N/A	0.51	0.80	0.24	0.36
	CM	**0.98**	**1.00**	0.13	**0.27**	**0.76**	1.00	0.30	**0.13**	0.88	**1.00**	**0.00**	0.33	**0.71**	**1.00**	0.03	0.28
	RP	0.29	**1.00**	0.34	0.39	0.48	1.00	0.53	0.42	**1.00**	**1.00**	0.19	N/A	0.63	**1.00**	0.33	0.34
CPU	DNN	0.85	1.00	**0.00**	0.33	0.80	**1.00**	0.09	0.43	**1.00**	**1.00**	**0.00**	N/A	0.05	0.05	0.35	**0.14**
	LLM	0.96	1.00	0.29	0.33	0.76	0.90	0.24	0.33	**1.00**	0.81	0.11	N/A	**1.00**	0.28	0.50	N/A
	CM	**1.00**	1.00	0.07	N/A	0.86	**1.00**	**0.05**	**0.11**	**1.00**	**1.00**	**0.00**	N/A	0.92	**1.00**	**0.09**	0.33
	RP	0.98	1.00	0.27	0.33	**0.88**	**1.00**	0.24	0.45	0.95	**1.00**	0.17	0.33	0.91	**1.00**	0.51	0.36
MEM	DNN	N/A	N/A	N/A	N/A	0.69	**1.00**	**0.23**	0.34	0.35	0.41	0.64	**0.33**	N/A	N/A	N/A	N/A
	LLM	N/A	N/A	N/A	N/A	0.78	0.99	0.43	0.37	0.00	0.00	N/A	N/A	N/A	N/A	N/A	N/A
	CM	N/A	N/A	N/A	N/A	0.70	**1.00**	0.28	**0.33**	0.38	0.65	0.65	**0.33**	N/A	N/A	N/A	N/A
	RP	N/A	N/A	N/A	N/A	**0.80**	**1.00**	0.51	0.39	**0.70**	**1.00**	**0.21**	0.36	N/A	N/A	N/A	N/A

P: *precision*, *R*: *recall*, *T/F*: *distance in the case of True/False positives*



(a) *Precision* RT (b) *Recall* CPU (c) *T* CPU (d) *F* RT

Fig. 5: RQ1: Critical differences plots

prompt (i.e. $K = 5$). The causal model is based on the dLiNGAM algorithm [20], [21], which is provided with the historical dataset and prior knowledge automatically derived as described in Section III.

VI. Results

A. RQ1: learning from data including past performance issues

The experimental results for answering RQ1 (for which the models learned from datasets containing examples of performance issues) are reported in Table I.

We conducted the Friedman test [22] with $alpha < 0.05$, which revealed significant statistical differences among models. We then performed the Nemenyi test and computed the critical differences (CD) for each combination. Figure 5 plots the critical differences per metric. Notably, CM emerges as the best model in terms of *precision*, for identifying RT and CPU issues, and of *recall* (along with RP). The only exception in *precision* is in detecting MEM issues, where RP is more effective. This mostly depends on the performance of CM on Online Boutique. The memory occupation of this subject exhibits significant instability. The historical dataset diverges considerably from the data obtained during test execution with the generated configurations. This discrepancy renders historical data-based approaches like CM less effective. In this scenario, RP, which generates random configurations independent of past data, emerges as the best choice. As for effectiveness, CM showed the best values of both *T* and *F* for CPU and RT issues, and the best value of *T* for RT issues; DNN is the best model for *F* on RT issues (no critical difference with CM).[6] The complete statistical results are available in the online repository.

As for µBench, CM has a *precision* two times that one of DNN and LLM in finding RT issues. For CPU, all models exhibit strong effectiveness because a small number of users leads CPU consumption to exceed the threshold. In our experiments, µBench does not exhibit memory issues in the workload space considered. Consequently, the models tend to not generate any "suspect" configurations for memory, avoiding unnecessary executions. This is achieved by all techniques but RP, which, as reported in Section V-B, generates 10 unnecessary tests (one per service).

As for SockShop, CM still presents a greater *precision* in finding RT issues. For MEM, all the models present high *precision* and *recall*, but the RP presents the best ones because the memory issues in this subject appear with a small number of users, therefore RP has a high probability of generating configurations with a user size large enough to produce a performance issue. However, RP presents a high value for the distance *T*, almost double CM and DNN values. In terms of CPU utilization, all models achieve strong *precision* scores. While RP exhibits the highest *precision* once again, it also demonstrates a significantly higher distance *T*. Conversely, CM achieves a distance close to zero, indicating its ability to identify the minimal configurations.

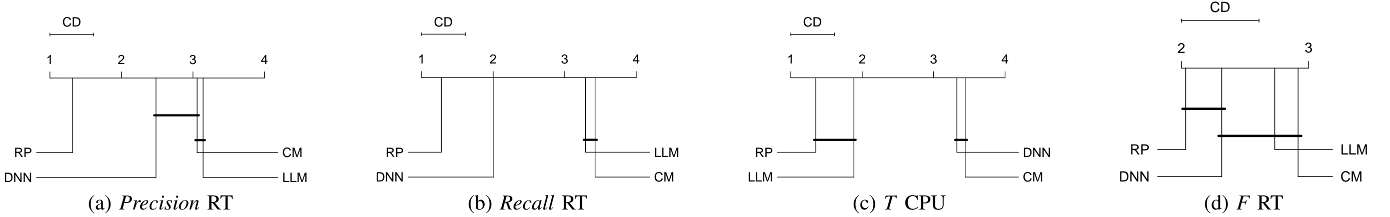
For Online Boutique, all the models present strong performance for RT. RP presents the maximum *precision* and

[6] For *T* and *F*, the ranking in the CD plot is inverted, with the first position representing the worst rank and the last position indicating the best rank.

TABLE II: RQ2: Models comparison (datasets without samples of past performance issues)

Performance metric	Model	µBench				SockShop				TeaStore			
		P	*R*	*T*	*F*	*P*	*R*	*T*	*F*	*P*	*R*	*T*	*F*
RT	DNN	0.24	**1.00**	**0.08**	0.34	0.09	0.52	0.49	**0.41**	0.06	0.50	**0.14**	0.31
	LLM	0.00	0.00	N/A	0.36	0.00	0.00	N/A	N/A	0.12	0.41	0.30	0.32
	CM	**0.55**	0.06	0.66	**0.33**	0.01	0.01	**0.40**	0.55	0.00	0.00	N/A	**0.17**
	RP	0.29	**1.00**	0.34	0.39	**0.48**	**1.00**	0.53	0.42	**0.63**	**1.00**	0.33	0.34
CPU	DNN	0.86	1.00	**0.00**	0.33	0.77	**1.00**	0.02	**0.16**	**1.00**	**1.00**	**0.00**	N/A
	LLM	**1.00**	1.00	0.27	N/A	**0.68**	0.89	0.01	0.47	0.49	**1.00**	0.48	**0.31**
	CM	0.91	1.00	**0.00**	0.33	0.78	0.99	**0.00**	0.33	0.14	0.85	**0.00**	0.48
	RP	0.98	1.00	0.27	0.33	0.88	**1.00**	0.24	0.45	0.91	**1.00**	0.51	0.36
MEM	DNN	N/A	N/A	N/A	N/A	0.70	**1.0**	**0.39**	0.46	N/A	N/A	N/A	N/A
	LLM	N/A	N/A	N/A	N/A	**0.83**	0.50	0.61	**0.33**	N/A	N/A	N/A	N/A
	CM	N/A	N/A	N/A	N/A	0.45	0.32	0.50	**0.33**	N/A	N/A	N/A	N/A
	RP	N/A	N/A	N/A	N/A	0.80	**1.00**	0.51	0.39	N/A	N/A	N/A	N/A

P: precision, R: recall, T/F: distance in the case of True/False positives



(a) *Precision* RT (b) *Recall* RT (c) *T* CPU (d) *F* RT

Fig. 6: RQ2: Critical differences plots

recall. Configurations with a small number of users lead response time to exceed the threshold. Therefore, RP has a higher probability of discovering issues than other models that overestimate RT, stopping prematurely the search. For memory issues, LLM and RP have good performance. LLM receives examples of configurations that lead to memory issues. These configurations are associated with a high number of users. As a result, the model 'learns' that configurations involving a large user size are likely to cause memory issues. For CPU, all the models present strong performance because a small number of users leads CPU consumption to exceed the threshold.

As for TeaStore, CM presents strong performance for *precision* and *recall* for RT. The DNN does not detect any response time issues. Also RP and LLM exhibit good *precision* and *recall*, but with a higher distance *T* with respect to CM, for which *T* is close to zero. For CPU and MEM, all models except DNN present high *precision* and *recall*.

It is finally worth to remark that all the presented precision, recall and distance values are obtained by running just the one test case (per service-performance metric pair) as suggested by the model, compared to the number of $|US| \times |LT| \times |SR|$ tests run with the ground truth manual strategy (Sec. V-B, Fig 4, number of users equal to 1).

B. RQ2: learning from data not including past issues

To answer RQ2, we consider historical datasets without examples of performance issues, obtained from the datasets used for RQ1 filtering out all entries containing at least one performance issue for a microservice on RT or MEM. Considering that CPU issues appear for a few users, we do not filter them as the resulting dataset would contain too few entries (about 40 for µBench, SockShop and Online Boutique). Applying the filtering criteria to Online Boutique yields a dataset that comprises solely executions involving a single user. For this reason, we restrict this analysis to the subjects µBench, SockShop, and TeaStore. We do not repeat the experiment for RP as it does not use the historical dataset. Table II shows the results of all models per subject and metric.

Like for RQ1, we employed the Friedman test to assess statistical differences among the models' evaluation metrics. The test revealed significant differences for all evaluation metrics. We also conducted the Nemenyi test and calculated critical differences. Figure 6 plots the critical differences for each evaluation metric. RP is the most effective model for identifying RT and CPU issues (the best in terms of *precision* and *recall*, except for CPU *recall* where it is the best along with DNN). In addition, RP and LLM are the best in terms of *precision* for MEM. Regarding efficiency, RP and DNN exhibited the best performance for *T* on RT and MEM. However, RP has the worst performance for *T* on CPU, where DNN and CM are the best ones. Finally, for *F*, the CR graphs do not exhibit critical difference among the models.

As for µBench, CM achieved the highest *precision* for RT issues, but a *recall* close to zero. Therefore, CM detects only some RT issues (generally it generates only one configuration). In contrast, the DNN model generates configurations for all services, but most do not expose issues. The LLM is not able to generate configurations for RT. Thus, for the RT, the best model for *precision* and *recall* is the RP. For memory, µBench does not expose any issues. CM and LLM do not generate any configurations, whereas the DNN model generates some configurations, which however are useless tests.

As for SockShop, all models present a *precision* close to zero in detecting RT issues, therefore any model detects actual response time issues. For the memory, LLM has the highest *precision* and, therefore, generates configurations with a high

TABLE III: RQ3: Time required by learning technique (seconds)

Model	µBench		SockShop		Online Boutique		TeaStore	
	RQ1	RQ2	RQ1	RQ2	RQ1	RQ2	RQ1	RQ2
	Mean (±Std)	Mean (±Std)	Mean (±Std)	Mean (±Std)	Mean (±Std)	Mean (±Std)	Mean (±Std)	Mean (±Std)
DNN	113.6 (±5.8)	109.8 (±5.1)	145.6 (±5.0)	76.8 (±7.6)	119.2 (±4.5)	N/A	94.9 (±4.5)	68.3 (±3.1)
LLM	248.8 (±0.6)	78.2 (±0.5)	316.8 (±1.7)	102.7 (±0.75)	185.6 (±2.0)	N/A	153.5 (±1.8)	71.7 (±0.9)
CM	3826.7 (±65.9)	4412.6 (±184.6)	5733.4 (±152.9)	8030.6 (±928.3)	3963.8 (±51.9)	N/A	2200.1 (±150.7)	2123.2 (±402.4)

accuracy. However, it does not present a high *recall*, thus, it does not detect all issues.

For TeaStore, all models present a *precision* close to zero on RT: they detect actual response time issues. For MEM, TeaStore does not expose issues. The DN and the LLM do not generate any configurations, whereas the DNN model generates some configurations; in this case, the DNN model generates useless tests.

To answer RQ1 and RQ2, we ran 5,562 tests. The tests were executed sequentially, each lasting 2 minutes with a 3-minute pause between consecutive tests. Excluding model training and configuration generation times, the experiment required approximately 464.5 hours (almost 20 days).

C. RQ3: comparative time analysis of learning strategies

For answering RQ3, we conduct a comparative analysis of the time required by each learning model for both model construction and configuration generation. Table III shows the detailed results. For CM, we measure the following time intervals: time to construct the causal graph (causal discovery), time for the fitting process, and time for the generation phase. Similarly, for DNN, we measure the training time of the neural network and the generation phase. Finally, for LLM we measure the time for context construction (i.e. embedding the historical dataset) and the generation phase. As before, all precision, recall and distance values are obtained by running just one test case (per service-performance metric pair), against $|US| \times |LT| \times |SR|$ tests for the highest effectiveness (namely, the ground truth manual strategy).

We replicated the statistical analyses employed in RQ1 and RQ2. This involved assessing the statistical differences in execution time among the models across two scenarios: when the historical dataset contains performance issues, and when not. The Friedman test revealed statistically significant differences in required time across all models in both scenarios. The results of the Nemenyi test, visualized as critical difference graphs, are presented in Figure 7. For RQ1, DNN is the fastest solution. For RQ2, the times required by DNN and LLM models are statistically similar. This can be attributed to the dominant time consumption by LLM during dataset embedding due to the context building. Since the dataset size is smaller in this scenario, the embedding time is reduced, leading to a total time comparable to DNN. CM exhibits significantly longer inference times (intervention time) compared to both DNN and LLM. This is true in RQ1 and RQ2. For CM, *the lack of GPU acceleration and non-optimized calculations contribute most significantly to the extended inference time*.

VII. Practical implications

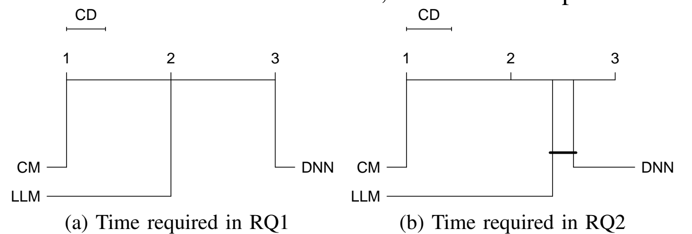
To draw general hints for performance testers, let us consider the following use cases:

1) The tester has a limited testing budget and needs to minimize the number of tests.
2) The tester needs to take action when operational conditions match the configuration generated by the model for performance issue prevention.
3) The tester has strict time constraints for training models.

For each use case, let us consider the further subcases where the dataset does not contain/does contain samples of performance issue (corresponding to RQ1 and RQ2, respectively).

For the first and second use cases, we rank the learning models using a weighted scoring system. First we associate an integer score to every model - the greater the score, the better the model. The score is based on the count $count_i$ of how many times the model is in position $i, i = 1, 2, 3$, with respect to other models for every performance metric and every subject - see Table I and Table II for the two mentioned subcases, respectively. To rank the models in the first use case, the metrics considered are *recall* and *F*. For the second use case, the scores for the ranking are based on *precision* and *T*. Consider, for instance, use case (1), subcase: dataset with samples of past performance issues. In Table I we see that DNN is the best model (position $i = 1$) for Recall R for metric RT for subject µBench, the best model for F for metrics RT and CPU for SockShop, and the best model for R for CPU for TeaStore, so its $count_1$ equals 4. The score of a model is calculated as the weighted sum $Score = 3count_1 + 2count_2 + count_3$. The final rank is based on the scores of the models. For the third use case, the ranking is determined based on the statistical tests in Figure 7.

Table IV reports the final rankings for each use case. For both the first and second use cases, CM is the best option when



(a) Time required in RQ1 (b) Time required in RQ2

Fig. 7: RQ3: Critical differences plots

TABLE IV: Ranking of learning models for the three performance testing use cases

Use case	Dataset	Ranking of models
Use case (1) High *recall* and low *F*	without issues	1. DNN, 2. RP, 3. CM, 4. LLM
	with issues	1. CM, 2. DNN, 3. RP, 4. LLM
Use case (2) High *precision* and low *T*	without issues	1. DNN, 2. RP, 3. CM, 4. LLM
	with issues	1. CM, 2. DNN, 3. RP, 4. LLM
Use case (3) Constrained training time	without issues	1. RP, 2. DNN, 2. LLM, 3. CM
	with issues	1. RP, 2. DNN, 3. LLM, 4. CM

historical datasets contain performance issues, while DNN is the best choice when they are absent.

Figure 8 presents the number of tests generated by each model. We differentiate between tests leading to performance issues (successful tests) and those that do not (unsuccessful tests). RP performs the worst for RQ1 as it generates three configurations for every service, including those aimed at discovering MEM issues in µBench and TeaStore, even though these subjects do not exhibit MEM issues. For RQ2, the differences between models are less pronounced, with RP however emerging as the best solution. This is because other models, lacking information about performance issues, tend to generate many false positives. Thus, a significant advantage of using models is their ability to refrain from generating configurations when there is no evidence suggesting a performance issue. This reduces costs compared to RP and manual methods. When historical data is not available, RP and manual are the only applicable approaches.

When time for training is constrained (third use case), RP is preferable, followed by DNN; CM shows a significant drop in ranking. The effectiveness of causal models is influenced by the early stage of currently available tools. However, the positive results suggest that better solutions can lead in the future to performance that may surpass ML techniques.

The performance of LLM is mainly influenced by the provided context. We queried it with a few-shot prompt; thus, the number of positive examples is vital. Insufficient issues in the historical dataset hinder the LLM's learning and result in poor performance. This is evident in the case of RT on SockShop, where the small number of positive examples delves into a less *precision*. In addition, the size of the LLM (i.e. the number of parameters) can impact performance. Smaller models might

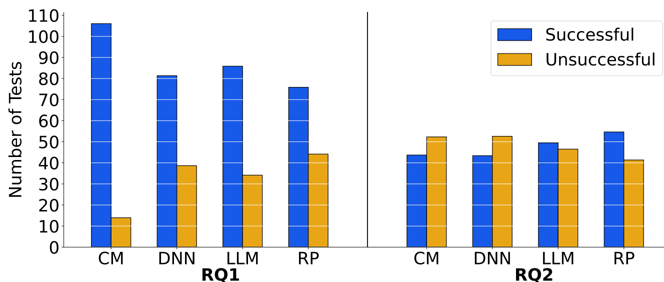


Fig. 8: Number of successful/unsuccessful tests per model

have limitations in handling complex data, and using a larger, more powerful model could improve results.

Manual and RP cost more in terms of the testing budget since they can run many unsuccessful tests. In particular, the former can run up to $9 \times (U_{max} - U_{min} + 1)$ tests for service-metric, and the latter always run one test for service-metric. Meaning that we cannot consider it when the budget is strict.

VIII. Threats to Validity

To qualify performance issues, we used scalability thresholds, which is a common approach in performance testing [13].

We assessed microWave efficacy using *precision* and *recall*, standard classifier metrics. To assess the efficiency, we defined a specific Hamming distance (D). To ensure that every configuration parameter has the same impact on the distance, we scaled the sub-distances (D_0, D_1, D_2) to a range of 0 to $\frac{1}{3}$; alternative distance definitions may yield different results.

The duration of the tests and their independence could impact the results of the experiment. Considering the substantial number of experiments conducted, we limited each test to 3 minutes and conducted the experiments sequentially. These choices introduce potential threats to the validity of the results. We allowed a 2-minute interval between tests to enable the systems to stabilize (CPU and memory return to nominal values), thereby mitigating the issue of test independence. In addition, each test was repeated five times for historical datasets and three times for model-generated tests.

Measurement errors could threaten internal validity. To mitigate this, all code was inspected, and experiments were replicated 20 times. The definition of load types may influence the results. Moreover, the generalizability of findings may be limited by the specific characteristics of the chosen subjects. Thus, a set of four diverse applications was selected.

All code and artifacts are openly available for verification and reproducibility at https://doi.org/10.5281/zenodo.14998802.

IX. Conclusions

Exposure of performance issues in microservices applications typically involves manual testing, resulting in significant time and resource consumption. We proposed a framework to use AI strategies to learn from past execution data to automatically generate workload configurations able to expose performance failures in microservices testing.

We analyzed the efficacy and the efficiency of three learning strategies – a Deep Neural Network, a Large Language Model, a Causal Model. The experimental results indicate that the causal model excels when past execution data contains instances of performance issues, while a neural network demonstrates superior performance in absence of such samples.

This comprehensive experimental study allowed to draw practical guidelines for practitioners to select the most suitable model for their specific needs.

REFERENCES

[1] Mingxuan Hui, Lu Wang, Hao Li, Ren Yang, Yuxin Song, Huiying Zhuang, Di Cu, and Qingshan Li. Unveiling the microservices testing methods, challenges, solutions, and solutions gaps: A systematic mapping study. *Journal of Systems and Software*, 220(112232):197–213, 2025.

[2] Siqi Shen, Vincent Van Beek, and Alexandru Iosup. Statistical characterization of business-critical workloads hosted in cloud datacenters. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 465–474. IEEE, 2015.

[3] Sen He, Glenna Manns, John Saunders, Wei Wang, Lori Pollock, and Mary Lou Soffa. A statistics-based performance testing methodology for cloud applications. In *Proceedings of the 2019 27th Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, page 188–199. ACM, 2019.

[4] Len Bass, Ingo Weber, and Liming Zhu. *DevOps - A Software Architect's Perspective*. Addison-Wesley, 2015.

[5] Robert Heinrich, André van Hoorn, Holger Knoche, Fei Li, Lucy Ellen Lwakatare, Claus Pahl, Stefano Schulte, and Johannes Wettinger. Performance engineering for microservices: Research challenges and directions. In *8th ACM/SPEC on International Conference on Performance Engineering Companion (ICPE '17 Companion)*, page 223–226. ACM, 2017.

[6] Alberto Avritzer, Vincenzo Ferme, Andrea Janes, Barbara Russo, Henning Schulz, and André van Hoorn. A quantitative approach for the assessment of microservice architecture deployment alternatives by automated performance testing. In Carlos E. Cuesta, David Garlan, and Jennifer Pérez, editors, *Software Architecture: 12th European Conference on Software Architecture*, pages 159–174, Cham, 2018. Springer International Publishing.

[7] Alberto Avritzer, Vincenzo Ferme, Andrea Janes, Barbara Russo, André van Hoorn, Henning Schulz, Daniel Menasché, and Vilc Rufino. Scalability assessment of microservice architecture deployment configurations: A domain-based approach leveraging operational profiles and load tests. *Journal of Systems and Software*, 165:1–16, 2020.

[8] Wellison R.M. Santos, Adalberto R. Sampaio Jr., Nelson S. Rosa, and George D.C. Cavalcanti. Microservices performance forecast using dynamic Multiple Predictor Systems. *Engineering Applications of Artificial Intelligence*, 129:2–16, 2024.

[9] Johannes Grohmann, Martin Straesser, Avi Chalbani, Simon Eismann, Yair Arian, Nikolas Herbst, Noam Peretz, and Samuel Kounev. SuanMing: Explainable Prediction of Performance Degradations in Microservice Applications. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering (ICPE)*, page 165–176. ACM, 2021.

[10] Aleksei Vasilevskii and Oleksandr Kachur. Self-service performance testing platform for autonomous development teams. In *Companion of the 15th ACM/SPEC International Conference on Performance Engineering (ICPE)*, page 242–248. ACM, 2024.

[11] Quinn Cooper, Diwakar Krishnamurthy, and Yasaman Amannejad. Budget aware performance test selection for microservices. In *2024 IEEE 17th International Conference on Cloud Computing (CLOUD)*, pages 376–385. IEEE, 2024.

[12] Matteo Camilli, Andrea Janes, and Barbara Russo. Automated test-based learning and verification of performance models for microservices systems. *Journal of Systems and Software*, 187:1–22, 2022.

[13] Matteo Camilli, Antonio Guerriero, Andrea Janes, Barbara Russo, and Stefano Russo. Microservices Integrated Performance and Reliability Testing. In *3rd International Conference on Automation of Software Test (AST)*, page 29–39. ACM, 2022.

[14] André de Camargo, Ivan Salvadori, Ronaldo dos Santos Mello, and Frank Siqueira. An architecture to automate performance tests on microservices. In *Proceedings of the 18th International Conference on Information Integration and Web-Based Applications and Services (iiWAS)*, page 422–429. ACM, 2016.

[15] Luca Giamattei, Antonio Guerriero, Ivano Malavolta, Cristian Mascia, Roberto Pietrantuono, and Stefano Russo. Identifying Performance Issues in Microservice Architectures through Causal Reasoning. In *Proceedings of the 5th ACM/IEEE International Conference on Automation of Software Test (AST)*, page 149–153. ACM, 2024.

[16] Alessio Zanga, Elif Ozkirimli, and Fabio Stella. A Survey on Causal Discovery: Theory and Practice. *International Journal of Approximate Reasoning*, 151:101–129, 2022.

[17] Judea Pearl. *Causality*. Cambridge University Press, 2 edition, 2009.

[18] Wengqi Fan, Yujuan Ding, Liangbo Ning, Shijie Wang, Hengyun Li, Dawei Yin, Tat-Seng Chua, and Qing Li. A Survey on RAG Meeting LLMs: Towards Retrieval-Augmented Large Language Models. In *KDD '24: Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 6491–6501, 2024.

[19] Marah Abdin, Sam Ade Jacobs, Ammar Ahmad Awan, et al. Phi-3 technical report: A highly capable language model locally on your phone, 2024.

[20] Shohei Shimizu, Patrik O. Hoyer, Aapo Hyvärinen, and Antti Kerminen. A linear non-gaussian acyclic model for causal discovery. *J. Mach. Learn. Res.*, 7:2003–2030, December 2006.

[21] Shohei Shimizu, Takanori Inazumi, Yasuhiro Sogawa, Aapo Hyvärinen, Yoshinobu Kawahara, Takashi Washio, Patrik O. Hoyer, and Kenneth Bollen. Directlingam: A direct method for learning a linear non-gaussian structural equation model. *J. Mach. Learn. Res.*, 12(null):1225–1248, July 2011.

[22] Borja Calvo and Guzmán Santafe. scmamp: Statistical Comparison of Multiple Algorithms in Multiple Problems. *The R Journal*, 8(1):248–256, 2015.