Microservices Performance Testing with Causality-enhanced Large Language Models

Cristian Mascia, Antonio Guerriero, Luca Giamattei, Roberto Pietrantuono, Stefano Russo
DIETI, Università degli Studi di Napoli Federico II, Napoli, Italy
{cristian.mascia, antonio.guerriero, luca.giamattei, roberto.pietrantuono, sterusso}@unina.it

***Abstract*—Efficient performance testing of microservices is essential for engineers to ensure that deviations of performance/resource usage metrics from expectations are promptly identified within their rapid release cycle. To this aim, engineers would need to explore the space of possible workload configurations and focus only on the critical ones, e.g., low-load configurations that unexpectedly cause performance issues. This requires a great effort, and can be infeasible in short release cycles.**

We present CALLMIT, a framework using Large Language Models (LLM) enhanced by causal reasoning to automatically generate critical workloads for microservices performance testing. Engineers query CALLMIT to generate workload configurations expected to expose deviations from performance requirements, so as to actually run only tests that trigger critical configurations. We present the experimental evaluation on three subjects, with comparison to a conventional Retrieval-Augmented Generation technique. The results show that causal models improve the correct identification by LLM of performance-critical workload configurations.

***Index Terms*—Microservices, Performance testing, Large Language Models, Causal reasoning, Retrieval-augmented generation**

I. Introduction

Performance testing of microservice applications is essential for understanding how workloads affect user experience and resources usage, for choosing among deployment alternatives, and to engineer proper mitigation means, like performance bug removal and capacity planning. In designing performance tests, engineers need to balance the goal of exposing performance issues with the stringent release deadlines typical of agile microservice development processes. As running tests for all possible workload configurations is unfeasible, the challenge is how to find *critical* configurations, expected to cause performance failures. Systems may exhibit performance failures, e.g., due to lack of robustness against heavy loads , or even at low load, due to other types of faults.

Existing microservices performance testing techniques and tools support the automation of tasks like tests execution [1], the deployment of test configurations given a manual specification in a Domain Specific Language [2], or the generation of tests with a workload inferred from the observed operational profile to mimic the expected usage [3]–[6]. We present CALLMIT (CAusality-enhanced Large Language model for MIcroservices performance Testing), a strategy to automate the generation of critical workload configurations and of the subsequent test cases, with the aim of saving testing effort by running only those ones more likely to lead to a degradation.

CALLMIT harnesses Causal Reasoning and Large Language Models (LLM). Specifically, we devise a new Retrieval Augmented Generation (RAG) strategy to improve the prompt to the `LLM` that uses the (automatically-inferred) causal relationships between the microservices' performance metrics to provide configurations more likely to produce a failure.

CALLMIT is experimentally evaluated in several variants, and compared to a conventional RAG technique on three popular subjects The results are that causal models can actually improve the ability of LLM to correctly identify performance-critical workload configurations.

All code and associated artifacts have been made openly available at https://github.com/uDEVOPS2020/CALLMIT

II. Related Work

Microservice performance testing. Microservices demand for *ad hoc* performance testing strategies with respect to other software architectures [7]. Vasilevskii *et al.* [8] highlight the challenges faced by performance engineers in companies, noting their reliance on manually developed in-house solutions. They advocate the establishment of shared practices and standardized tools to address the challenges.

Not many solutions have been proposed for automating performance testing of microservices. Most of them leverage past executions data for tests generation 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.

Camilli *et al.* [6] employ probabilistic model checking to identify performance issues in microservices and correlate them to system-level requirements. They construct a Continuous Time Markov Chain from load testing data, enabling automated verification and the calculation of a score to evaluate deployment alternatives.

The same authors propose a methodology for microservices joint performance and reliability testing [5]. By leveraging operational data, it partially automates tests at decision gates of DevOps processes, providing better insights compared to separate performance and reliability testing.

Avritzer *et al.* [3], [4] evaluate the scalability of deployment configurations through load testing based on operational profiles derived from past executions data. Tests assess performance (e.g., response time) and configurations scalability under an operational profile with increasing loads.

LLM for testing. Advances in LLM have transformed various domains, including software testing. Wang *et al.* [9] provide

a comprehensive review on LLM-based software testing, in tasks like unit tests generation, system tests generation, test oracle generation, program repair, debugging, and bug analysis, discussing the current gaps and research directions.

Santos *et al.* [10] demonstrated the potential of LLM in automating test case generation, facilitating debugging, and supporting testers with coding tasks. Their study, conducted in industrial settings, reveal that while LLM enhance testing practices, their adoption necessitates caution and the establishment of robust guidelines to ensure proper utilization.

We propose leveraging LLM in combination with a *causal graph* to automate the generation of workload configurations likely to reveal performance issues in microservices performance testing. We introduce the CALLMIT framework, and we evaluate two LLMs and four ways of integrating a causal graph within them, using three subjects widely adopted in microservice research [4], [6], [11], [12].

III. The proposed framework

A. Overview

Figure 1 shows the architecture of the proposed framework. CALLMIT exploits an LLM augmented with a causal graph for detecting workload configurations that lead to performance issues in a microservices application. A performance issue is defined as any instance where one or more observed performance metrics exceed a specified threshold during a testing or monitoring session.[1] A workload configuration is defined as a triple <*User size*, *Load type*, *Spawn rate*>, where:

- *User size (US)*: number of concurrent users;
- *Load type (LT)*: operational profile, specified categorically (e.g., uniform, unbalanced towards some services);
- *Spawn rate (SR)*: number of users to spawn per second.

CALLMIT uses a dataset obtained by monitoring past executions (as usual in microservice applications) in order to guide the generation of workload configurations. As in past works [13], we build the dataset from monitoring data (logs and performance metrics), inferring the workload configuration parameters (WP) (*US, LT, SR*) and the resource usage (i.e. CPU and memory usage), response time (RT) and request rate for each service. The data is used to augment the prompt given to the LLM with two items, namely a context and a set of causal paths. As for the former, historical data is used like in a conventional RAG system (Retrieval-Augmented Generator in Figure 1), providing the LLM with contextual data. As for the latter, past data is used to build a causal graph, which is given to the LLM in the form of causal paths to improve its outputs (Causal Paths Retriever in Figure 1).

B. Retrieval-Augmented Generator

CALLMIT leverages Retrieval-Augmented Generation for the purpose of supplying the LLM with contextual data. This requires to specify *how the historical dataset is partitioned* (chunking), *which embedding model is used*, and *how the chunks to be provided to the prompt are selected*. To preserve the semantics of data in the dataset, the latter is partitioned by rows, with each row serving as an individual chunk. The chunks and the question are then converted into a vector through a pretrained embedding model (all-MiniLM-L6-v2-f16). The best chunks to retrieve are identified by running a similarity search (cosine similarity) with the question.

C. Causal Paths Retriever

The Causal Paths Retriever (CPR) enhances the LLM by augmenting the prompt with causal knowledge.

Let a weighted directed acyclic graph (DAG) $\mathcal{G} = (\boldsymbol{X}, \mathcal{E}, \mathcal{W})$ be the *causal graph* representing a linear causal model – specifically a Structural Equation Model (SEM) - where nodes $X_i \in \boldsymbol{X}$ are random variables, edges $e_i \in \mathcal{E}$ denote the causal relationships between them, and $\mathcal{W}$ is an $|\boldsymbol{X}| \times |\boldsymbol{X}|$ adjacency matrix $\mathcal{W} = \{w_{i,j}\}$, with $w_{i,j}$ representing the connection strength from X_i to X_j. CALLMIT learns this model automatically from the historical dataset via a causal discovery algorithm, that is `dLiNGAM` [14]. For each service, the dataset has four variables for the considered performance metrics (request rate, RT, CPU usage, and memory consumption), and three variables representing the workload parameters WP (user size, load type, spawn rate).

The causal model is meant to represent the relations between the WP and the performance metrics of interest (e.g., RT, memory usage) for all the services. To prompt the LLM, we consider a set of causal paths in the graph. A causal path $\mathcal{P} = X_0 \rightarrow X_1 \rightarrow \cdots \rightarrow X_n$ is a finite directed path, namely a sequence of (non-repeated) edges $(e_1, e_2, \ldots, e_{n-1}) \in \mathcal{E}$ directed in the same direction that joins a sequence of distinct vertices $(X_1, X_2, \ldots X_n) \in \boldsymbol{X}$. The task of the CPR is to get the most informative paths that link the WP to the performance metric of interest, so as to highlight what changes in the WP is more likely to affect that metric, net of possible confounders.

Since the number of nodes and edges can increase significantly as the number of services grows, and since the performance of LLM can decrease with long prompts [15], CALLMIT selects a subset of causal paths. These need to be *workload-related causal paths*, i.e. paths originating from the WP nodes (i.e., US, LT and SR) and leading to the target node of interest (i.e., a node representing the service-metric pair for which we want to generate a critical configuration, e.g., response time of service S_i.

In order to reduce the number of selected paths and focus on the most impacting ones, we exploit the notion of *causal strength* - an estimate of how much the effect variable is expected to change for a change in the cause variable [16] - in the form of *edge strength* (or connection strength) coefficients $\mathcal{W}w_{i,j}$ as computed by `dLiNGAM` [14] based on the covariance matrix. The *strength of a causal path* is the sum of the causal strengths of its constituent edges.

We use the strength to select only the top-k paths. In the experimentation, we test two variants of CALLMIT, top-1 and top-5. Moreover, in order to reduce the time to search for the strongest paths, we consider a *pruned* causal graph,

[1]The thresholds can be either defined manually or inferred automatically as in Avritzer *et al.* [4].

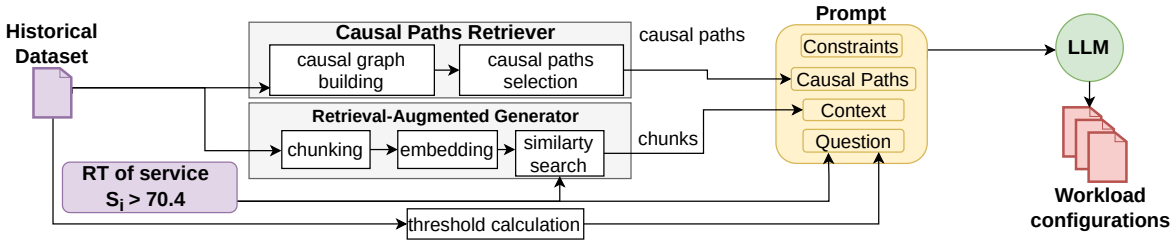

Fig. 1. CALLMIT architecture

i.e., a graph in which several edges are removed, based on a *causal strength threshold* – all the edges with a strength less than a causal strength threshold Th_{CS} (set to 0.3 during experimentation) are removed from the graph, resulting in a new graph $\mathcal{G}' = (\boldsymbol{X}, \mathcal{E}', \mathcal{W}')$ where $\mathcal{E}' = \{e \in \mathcal{E} \mid |\mathcal{W}'(e)| > Th_{CS}\}$. This gives four variants of CALLMIT (top-1 and top-5, with and without pruning).

The prompt is composed of the following sections:

- *Generation Constraints*: the constraints specify the accepted values for WP, i.e. the boundaries for the *User size*, the categories for the *Load type*, and the acceptable ranges for *Spawn rate*;
- *Causal Paths*: the causal paths selected;
- *Contextual Data*: the context is provided by the Retrieval-Augmented Generator;
- *Question*: the question invokes the LLM generation, specifying the target metric and the thresholds.

Thresholds for establishing whether a configuration is critical or not are defined according to Avritzer *et al.* [4] (*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. Experimentation

We evaluate the benefit in detecting performance issues of augmenting the LLM prompt with causal paths. The experiments use three subjects and two LLM. These are queried to detect which workload configurations result in performance issues, by varying the user size *US* – a natural number, which we assume bounded in $[U_{min}, U_{max}]$ - the load type *LT* – which we assume to be categorical variables in the experimentation - and the spawn rate *SR*.

As evaluation metrics, we use *precision*, number of performance issues correctly detected over number of all potential performance issues predicted by the model; *recall*, number of issues correctly detected over number of actual issues; F_1, which is their armonic mean.

A. Subjects

`µBench`[2] [17] is an artificial microservice application composed of 10 services, with a random service dependency graph and a random stressing function per service (stress/idle function). `SockShop`[3] is a benchmarking application for cloud-native microservices testing, emulating an e-commerce site; it is composed of 8 services (6 edge services). `TeaStore`[4] [18] is a benchmark emulating a web store with automated customer orders, comprising a registry and five services.

B. Baseline and CALLMIT variants

As baseline, we use the LLM simply integrated with a RAG system (called `RAG`) and compare it against four variants of CALLMIT, which enhances `RAG` with the causal paths retriever. We consider these variants for paths selection:

- **S** top-1: It selects the **S**trongest *causal path* (one for each workload parameter);
- **S**$_5$ top-5: It selects the five **S**trongest *causal path* (five for each workload parameter);
- **SP** top-1 with pruning: It selects the **S**trongest *causal path* (one for each workload parameter) on the **P**runed graph;
- **SP**$_5$ top-5 with pruning: It selects the five **S**trongest *causal path* (five for each parameter) on the **P**runed graph;

C. Experiments Setup

For the experiments, we have synthesized a historical dataset for each subject. Specifically, we considered: all values in $(U_{max} - U_{min})$ for the user size *US*; three values for the load type *LT* – one for balanced workload (every service receives the same amount of requests), and two with a workload unbalanced toward a subset of services simulating two use cases (CPU- and memory-intensive, respectively) - three values for spawn rate *SR* {1,5,10}. Each workload configuration lasts three minutes and is run five times, with a two-minute pause between runs to avoid carryover effects (consecutive runs influencing each other, e.g., in CPU and memory usage).

To construct the causal graph, we provide the `dLiNGAM` *causal structure discovery* algorithm with the historical dataset and prior knowledge. Prior knowledge is specified as required or forbidden edges. We defined the edges connecting *WP* to the request rate for each service as mandatory.

Finally, considering the LLM scaling law outlined in [19], we opted for two LLM with a remarkably difference in size: `phi3.5` [20] and `gemini-1.5-flash` (`gemini`) [21].

D. Results

Table I reports the results (average over 10 repetitions) with the Dunn's test results comparing each variant *vs* RAG ($\alpha = .05$) [22]. Results about memory for `SockShop` are in Table II, as `µBench` and `TeaStore` did not exhibit memory issues.

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

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

[4]https://github.com/DescartesResearch/TeaStore-

TABLE I
RESULTS IN DETECTING RESPONSE TIME AND CPU ISSUES. UNDERLINED: BEST AVG. VALUES. BOLDFACE: SIGNIFICANTLY DIFFERENT FROM RAG

LLM	Performance metric	Technique	µBench			TeaStore			SockShop		
			precision	*recall*	F_1	*precision*	*recall*	F_1	*precision*	*recall*	F_1
phi3.5	RT	S	0.490	1.000	0.657	**0.782**	0.902	**0.837**	0.462	1.000	0.632
		S_5	0.580	1.000	0.718	0.667	1.000	**0.800**	0.462	1.000	0.632
		SP	0.300	1.000	0.458	**0.940**	**0.576**	**0.700**	0.462	1.000	0.632
		SP_5	**0.200**	1.000	**0.333**	0.667	**0.571**	0.615	0.462	1.000	0.632
		RAG	0.430	1.000	0.599	0.333	1.000	0.500	0.462	1.000	0.632
	CPU	S	**1.000**	**1.000**	**1.000**	**0.450**	**0.112**	0.180	**0.846**	1.000	**0.917**
		S_5	**1.000**	**0.900**	**0.945**	**0.500**	**0.125**	0.200	**0.846**	1.000	**0.917**
		SP	**1.000**	0.890	**0.940**	**0.500**	**0.125**	0.200	**0.846**	1.000	**0.917**
		SP_5	**1.000**	**1.000**	**1.000**	**0.500**	**0.125**	0.200	**0.846**	1.000	**0.917**
		RAG	0.870	0.756	0.807	0.134	0.417	0.200	0.815	1.000	0.898
gemini	RT	S	0.720	1.000	0.837	1.000	1.000	1.000	**0.500**	1.000	**0.666**
		S_5	**0.770**	1.000	**0.869**	1.000	1.000	1.000	**0.454**	1.000	**0.621**
		SP	0.690	1.000	0.816	1.000	1.000	1.000	**0.500**	1.000	**0.666**
		SP_5	0.700	1.000	0.824	1.000	1.000	1.000	**0.385**	1.000	**0.556**
		RAG	0.680	1.000	0.807	1.000	1.000	1.000	0.308	1.000	0.471
	CPU	S	1.000	1.000	1.000	1.000	**1.000**	**1.000**	**1.000**	1.000	**1.000**
		S_5	1.000	1.000	1.000	1.000	**1.000**	**1.000**	**1.000**	1.000	**1.000**
		SP	1.000	1.000	1.000	1.000	**1.000**	**1.000**	**1.000**	1.000	**1.000**
		SP_5	1.000	1.000	1.000	1.000	**1.000**	**1.000**	**1.000**	1.000	**1.000**
		RAG	1.000	1.000	1.000	1.000	0.878	0.935	0.923	1.000	0.960

Does the causal path retriever improve performance?: CPR shows superior performance compared to RAG, achieving consistent improvements across all metrics. On average, CPR outperforms the baseline by 6.5% in F_1 score. This improvement is primarily driven by a 13.1% increase in *precision*, despite a slight decrease in *recall* (1.68%). The *recall* reduction is primarily because RAG, which does not exploit any kind of knowledge, tends to always generate some configuration without specifically looking for critical ones (hence, its low precision) – this happens mainly with TeaStore, where the large reduction impacts the average, while *recall* is 1 in most of the other cases.

On µBench, CPR achieves a consistent improvement, with a gain of 4.45%, 5.1%, and 4.24% in *precision*, *recall*, and F_1 score, respectively. On TeaStore, CPR yields a larger improvement, 31.8%, in *precision* but with a decrease in *recall* (12.47%). Nonetheless, the F_1 score improves by 11.31%. On SockShop, where both CPR and the baseline maintain a *recall* of 1, the performance improvement is measurable in terms of *precision*, which increases by 7.95%.

Does the causal strength selection strategy improve performance?: We compare the selection strategies (S, S_5) against RAG. On average, the two strategies enhance performance by 13.73% in *precision*, 8.44% in F_1 score, with a slight decrease in *recall*. The largest impact is on *precision*, consistently with the observation that RAG exhibits high *recall*.

Does pruning impact performance?: We compare the two pruning strategies (SP and SP_5) against RAG. On average, they yield a 4.56% increase in F_1 score, which is about half the improvement of non-pruning techniques. This reduced gain is due to a 3.12% decrease in *recall*. We conclude that pruning trades the reduced paths retrieval time off with performance.

How much does CALLMIT benefit different LLM?: For gemini, the baseline performance is already high, so augmenting the LLM prompt with CPR has a lower impact. Nonetheless, CALLMIT improves performance on average by 5.65%, 1.77%, and 5.33% in *precision*, *recall*, and F_1 score, respectively. For phi3.5, the improvements are more pronounced, expecially for F_1, which increases by 8.09%. This gain is driven by the improvement in *precision* (24.51%), paid in terms of *recall* (5.53% decrease). Applying CALLMIT to a larger LLM results in a less pronounced but consistent improvement across the metrics.

TABLE II
RESULTS IN DETECTING MEMORY ISSUES ON SOCKSHOP

Technique	phi3.5			gemini		
	precision	*recall*	F_1	*precision*	*recall*	F_1
S	0.569	1.000	0.724	0.815	1.000	0.898
S_5	0.638	1.000	0.779	0.815	1.000	0.898
SP	**0.785**	1.000	**0.879**	0.808	1.000	0.893
SP_5	0.692	1.000	0.818	**0.846**	1.000	**0.917**
RAG	0.669	1.000	0.800	0.769	1.000	0.870

V. CONCLUSIONS

Detecting performance issues in microservices applications often relies on manual testing, which is both time-consuming and resource-intensive. We proposed a framework that integrates a Large Language Model with a Causal Graph, enabling the automatic generation - based on past execution data - of workload configurations able to expose performance failure.

We evaluated the benefit of the Causal Graph by comparing two strategies for causal path selection. Experimental results demonstrate that incorporating a Causal Graph enhances the LLM performance in identifying performance issues.

ACKNOWLEDGMENT

This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 871342 "uDEVOPS".

REFERENCES

[1] A. de Camargo, I. Salvadori, R. Mello dos Santos, and F. 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.

[2] V. Ferme and C. Pautasso. A Declarative Approach for Performance Tests Execution in Continuous Software Development Environments. In *ACM/SPEC International Conference on Performance Engineering*, page 261–272. ACM, 2018.

[3] A. Avritzer, V. Ferme, A. Janes, B. Russo, H. Schulz, and A. van Hoorn. A quantitative approach for the assessment of microservice architecture deployment alternatives by automated performance testing. In C. E. Cuesta, D. Garlan, and J. Pérez, editors, *Software Architecture*, pages 159–174, Cham, 2018. Springer International Publishing.

[4] A. Avritzer, V. Ferme, A. Janes, B. Russo, A. van Hoorn, H. Schulz, D. Menasché, and V. 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.

[5] M. Camilli, A. Guerriero, A. Janes, B. Russo, and S. Russo. Microservices Integrated Performance and Reliability Testing. In *3rd International Conference on Automation of Software Test (AST)*, page 29–39. ACM, 2022.

[6] M. Camilli, A. Janes, and B. Russo. Automated test-based learning and verification of performance models for microservices systems. *Journal of Systems and Software*, 187:1–22, 2022.

[7] R. Heinrich, A. van Hoorn, H. Knoche, F. Li, L. E. Lwakatare, C. Pahl, S. Schulte, and J. 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.

[8] A. Vasilevskii and O. 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.

[9] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang. Software testing with large language models: Survey, landscape, and vision, 2024.

[10] R. Santos, I. Santos, C. Magalhaes, and R. de Souza Santos. Are We Testing or Being Tested? Exploring the Practical Applications of Large Language Models in Software Testing. In *2024 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 353–360. IEEE, 2024.

[11] M.S. Floroiu1, S. Russo, L. Giamattei, A. Guerriero, I. Malavolta, and R. Pietrantuono. Anomaly Detection and Root Cause Analysis of Microservices Energy Consumption. In *2024 IEEE International Conference on Web Services (ICWS)*, page 580–590. IEEE, 2024.

[12] W. Meijer, C. Trubiani, and A. Aleti. Experimental evaluation of architectural software performance design patterns in microservices. *Journal of Systems and Software*, 218:1–15, 2024.

[13] L. Giamattei, A. Guerriero, I. Malavolta, R. Pietrantuono C. Mascia, and S. Russo. Identifying performance issues in microservice architectures through causal reasoning. In *2024 IEEE/ACM International Conference on Automation of Software Test (AST)*, pages 149–153, 2024.

[14] S. Shimizu, T. Inazumi, Y. Sogawa, A. Hyvärinen, Y. Kawahara, T. Washio, P. O. Hoyer, and K. Bollen. DirectLiNGAM: A Direct Method for Learning a Linear Non-Gaussian Structural Equation Model. *Journal of Machine Learning Research*, 12:1225–1248, jul 2011.

[15] M. Levy, A. Jacoby, and Y. Goldberg. Same Task, More Tokens: the Impact of Input Length on the Reasoning Performance of Large Language Models. In L.-. Ku, A. Martins, and V. Srikumar, editors, *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 15339–15353, Bangkok, Thailand, 2024. Association for Computational Linguistics.

[16] D. Janzing, D. Balduzzi, M. Grosse-Wentrup, and B. Schölkopf. Quantifying causal influences. *The Annals of Statistics*, 41(5):2324–2358, 2013.

[17] A. Detti, L. Funari, and L. Petrucci. μBench: An Open-Source Factory of Benchmark Microservice Applications. *IEEE Transactions on Parallel and Distributed Systems*, 34(3):968–980, 2023.

[18] J. v. Kistowski, S. Eismann, J. Grohmann, N. Schmitt, A. Bauer, and S. Kounev. TeaStore - a micro-service reference application for performance engineers. In *Companion of the 2019 ACM/SPEC International Conference on Performance Engineering (ICPE)*, pages 47–48. ACM, 2019.

[19] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei. Scaling laws for neural language models, 2020.

[20] M. Abdin et al. Phi-3 technical report: A highly capable language model locally on your phone, 2024.

[21] Gemini Team. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context, 2024.

[22] Olive Jean Dunn. Multiple comparisons using rank sums. *Technometrics*, 6(3):241–252, 1964.