# Generating Representative Macrobenchmark Microservice Systems from Distributed Traces with Palette

Vaastav Anand
Max Planck Institute for Software Systems
Saarbrücken, Germany

Matheus Stolet
Max Planck Institute for Software Systems
Saarbrücken, Germany

Jonathan Mace
Microsoft Research
Seattle, USA

Antoine Kaufmann
Max Planck Institute for Software Systems
Saarbrücken, Germany

## Abstract

Microservices are the dominant design for developing cloud systems today. Advancements for microservice need to be evaluated in representative systems, e.g. with matching scale, topology, and execution patterns. Unfortunately in practice, researchers and practitioners alike often do not have access to representative systems. Thus they have to resort to sub-optimal non-representative alternatives, e.g. small and oversimplified synthetic benchmark systems or simulated system models instead.

We propose to solve this issue using distributed trace datasets, available from large internet companies, to generate representative microservice systems. To do so, introduce a novel abstraction of a *system topology* which uses Graphical Causal Models (GCMs) to model the underlying system by incorporating branching probabilities, execution order of outgoing calls to every dependency, and execution times. We then incorporate this topology in Palette, a system that generates representative flexible macrobenchmarks microservice systems from distributed traces.

## CCS Concepts

• **Networks** → **Cloud computing**; • **Software and its engineering** → **Cloud computing**.

## Keywords

Microservices, Distributed Tracing, Microservice Benchmarks



*APSys '25, Seoul, Republic of Korea*



ACM ISBN 979-8-4007-1572-3/2025/10
https://doi.org/10.1145/3725783.3764387

**ACM Reference Format:**
Vaastav Anand, Matheus Stolet, Jonathan Mace, and Antoine Kaufmann. 2025. Generating Representative Macrobenchmark Microservice Systems from Distributed Traces with Palette. In *16th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '25), October 12–13, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3725783.3764387

## 1 Introduction

Modern cloud systems are developed as microservice systems. They have been adopted by many large companies such as Facebook [16], Netflix [6], Uber [14], among others [5, 15] since the enable individual services to be developed, deployed, and scaled independently.

Validating and testing new advancements for microservices often requires developers to experiment with interventions—that is change aspects of the system with new design decisions, architectural choices, algorithms, backend components, and other strategies—in order to estimate and assess their impact on the system.

Ideally, developers and researchers would execute these intervention experiments on production systems to make claims about the technique's scalability and generalizability. However, access to production systems is limited to a select few. Even if one could procure access, the scope of experimentation is limited to minimize disruptions.

Despite the limited access to production systems, distributed traces are often readily available and researchers can use them to glean insight into these systems. Distributed traces capture rich structural and temporal information about the execution of the system, such as latency, execution patterns, branch probabilities, and call probabilities.

*We posit that distributed traces can support general purpose intervention experiments given the rich volume of system behavior they capture.*

However, we believe that currently there are three key challenges that prevent distributed traces from being used for intervention experiments. First, while distributed traces

capture a variety of metrics, they are observations of the intrinsic behavior of the system. The system behavior that caused these observations is not explicitly captured in distributed traces. Thus, we need a mechanism that can use the distributed traces to derive a model of the intrinsic behavior of the system. Second, currently there is no mechanism to convert intervention experiments into targeted modifications of the system without modifying the rest of the system behavior. The fundamental tenet of an intervention experiment requires that all the other factors in the system must be held constant to isolate the impact of the intervention. Third, converting distributed traces to a runnable system for supporting general purpose intervention experiments is non-trivial as different intervention experiments might wish to preserve different characteristics of the systems.

To overcome these challenges, we propose Palette, a system designed for natively supporting intervention experiments using distributed traces. Palette provides a new abstraction called *system topology* that models system behavior captured by generating a graphical causal model (GCM). To support targeted modifications for interventions, Palette provides a set of primitive operations that allow developers to make localized changes in the system topology for a given intervention. Finally, Palette provides a generation mechanism for converting a system topology to a runnable system. Our solution leverages the GCM at runtime to model the causal dependencies from the original system and uses it to more faithfully sample metrics for execution behavior, such as the amount of work to be done in a service or the payload size to be used between two services given the current state of the system.

# 2 Background and Motivation

## 2.1 Research Use Cases

Microservices have a large design space owing to its hetereogeneous nature. Consequently, the set of all possible interventions a researcher can perform is also large. For instance, researchers building a new network stack may be interested in confirming that their design maintains low tail latency under load so that requests meet their SLOs. To test their hypothesis they need a system that matches the characteristics of a real system such as request size, service execution times, and call graph depth and width. Request size matters because systems optimize differently based on its average. Service execution is critical because services with long execution times may see negligible benefits from a faster network stack. Similarly, benefits may be amplified in deep call graphs or offset by slow services in the critical path. Therefore, generated systems should be able to modify and preserve these properties so that interventions can be added while still maintaining realistic performance characteristics.

| Use-case | Meaningful properties |
|---|---|
| $\mu$Service Performance[33, 34, 37] | Varying graph sizes, execution paths, topology |
| Network Stacks[12, 19, 38] | Request sizes, service execution time, call depth, and call width |
| Congestion Control[21, 43] | Varying topology, timeout values |
| Res. Management[27, 32] | Varying execution paths, large service graphs |
| Tracing Framework[11, 22, 35, 39] | Varying execution paths, large number of services |
| RPC Framework [18, 20] | Varying request sizes |

**Table 1: Research use-cases and the ideal properties from a benchmark system.**

Different use-cases, as evidenced in Table 1, will care about different properties when introducing interventions to the system. No single point solution system derived from traces can support all possible interventions correctly.

## 2.2 The Role of Distributed Tracing

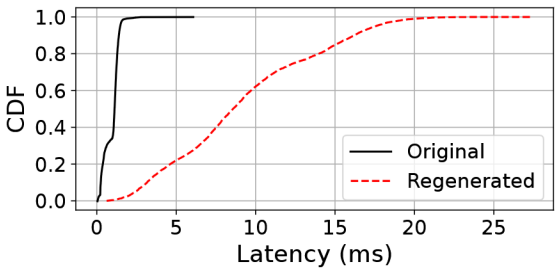
Distributed tracing is a critical monitoring component in modern cloud systems. It provides troubleshooting support for developers and operators during incident root-cause analysis and post-mortems. Distributed tracing supports this by generating an execution trace of each request across all components of the system. Additionally, anonymized traces have been one of the few ways for large-scale system operators to share meaningful details about their systems and workloads [23, 24]. We thus believe that the data-rich nature of distributed trace datasets represents an opportunity for addressing the structural diversity issue in existing open-source microservice systems for research use-cases.

**Trace Details.** A distributed trace contains the partially-ordered list of all APIs (referred to as spans) executed by the system to service the request. For each span, the tracing framework captures the total amount of execution time and the specific service (or component) at which the span was executed. The tracing frameworks encodes caller-callee relationships between APIs as parent-child relationships.

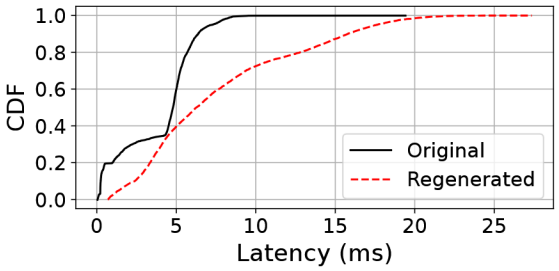
**Distributed Tracing Workload Generation.** Distributed tracing can also be used to generate realistic workloads to test solutions at large scale because of its rich collection of execution data [9, 28].

## 2.3 Graphical Causal Models (GCMs)

Graphical Causal Models [10] are directed acyclic graphs which encode causal relationships between the different nodes in the graph. Each node represents a variable, i.e. some observable data, and the edges represent causality. A directed




(a) Service A




(b) Service C

**Figure 1: Latency CDF mismatch with simple statistical approach**

edge between two nodes signifies that a variable influences the value of the other variable.

These models are useful for understanding the causal effects of one variable on another [3] and have been extensively used in root cause analysis of microservice systems [4, 8, 17, 36, 42]. GCMs help preserve the properties of the original system an can generate new samples (i.e. traces) based on the causal structure of the system. As GCMs internally use inferred causal effects from the observed data, any new sample generated by the GCM will be representative of the original data. Due to this representativeness retention property GCMs have been used in conjunction with traces for performing intervention experiments in simulation [1, 40].

## 2.4 Existing Approaches

**Trace Replay.** A system that replays exact traces generated from the original system guarantees that system behavior will be representative. However, the issue arises when a researcher performs an intervention and changes some component or aspect of the system. In that situation, the system will no longer have the ability to replay a prior execution because that execution is not present in the available traces. As a result, the system could diverge from the original system beyond the true impact of the intervention.

**Simple Statistical Models.** Another possible strawman solution is to calculate an aggregate statistic (e.g. mean) for each API as well as the execution probability for each downstream dependency call. However, this approach fails when a downstream API has multiple callers each of which is providing a different amount of work. Let's consider the scenario with three services, where both Service A and Service C make calls to a downstream Service B to perform some amount of work. Service C performs an order of magnitude higher work than Service A which results in Service C latency to be an order of magnitude higher than Service A. However, when we collect statistics about the downstream Service B, this information is lost as we only get 1 statistic which is the mean. If we were to re-generate the system using just this simple statistic, then we would find that Service B would almost take the same amount of time for calls originating from Service C but take almost an order of magnitude higher amount of time for calls originating from Service A. Figure 1 shows exactly this scenario where we find that the latency distribution of Service A is shifted by almost an order of magnitude. The reason why this approach fails is because the latency of service B does not condition the sampling of the latency on the upstream caller.

**Open-source microservice benchmarks.** These open-source systems are single point solutions in the large design space of microservice systems. While these systems [13, 26, 44] are useful targets for validation and good targets for some use-cases, the systems do not cover a representative enough design space for researchers to explore [30]. Thus, these are only useful for the specific use cases which do not require characteristics from the design space these systems do not represent. For example, performing an intervention experiment to test the efficacy of a new distributed tracing system requires the system to preserve the property of a large number of services and large fan-ins and fan-outs that are commonly seen in production systems [16, 23, 24, 29, 41]. However, this is not possible with existing open source systems as even the largest available open-source microservice system, TrainTicket [44], has only 45 services excluding caches and databases. Researchers may instead opt for synthetic benchmark generators such as $\mu$Bench [7] or Microsim [31] for some of their experiments but these synthetic generators often lack the representativeness and flexibility to truly support all possible intervention experiments that the researchers might wish to perform.

# 3 Palette Design

Figure 2 shows the design pipeline of Palette. Palette processes trace datasets to generate a system topology that encodes structural relationships, execution patterns, and performance characteristics of the observed system. This topology is converted into specifications that encodes mechanisms to ensure the preservation of the learned characteristics. Palette

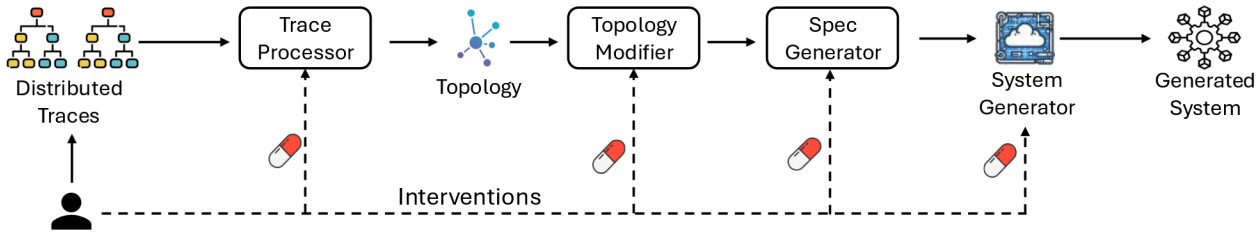



**Figure 2: Palette Pipeline**

| Behavior | Causal Equation |
|---|---|
| Probability($c$) | $p_a \sim Bernoulli(c)$ |
| Sequential($a_1, \ldots, a_n$) | $p_{a_1} * \lambda_{a_1} * a_1 + \cdots + p_{a_n} * \lambda_{a_n} * a_n + C$ |
| Concurrent($a_1, \ldots, a_n$) | $max(p_{a_1} * \lambda_{a_1} * a_1, \ldots, p_{a_n} * \lambda_{a_n} * a_n) + C$ |
| Choice($a_1, \ldots, a_n$) | $p_{a_1} * \lambda_{a_1} * a_1 + \cdots + p_{a_n} * \lambda_{a_n} * a_n + C$, such that, $p_{a_1} + \cdots + p_{a_n} = 1$ |

**Table 2: Causal Equation for modeling latencies used for a given GCM node where $a_i$ is the latency for API $i$, $\lambda_{a_i}$ is the coefficient in the GCM model for $a_i$, $C$ is the intercept extracted from fitting the GCM linear equation, and $p_{a_i}$ is a value of 0 or 1 signalling if API $i$ was called.**

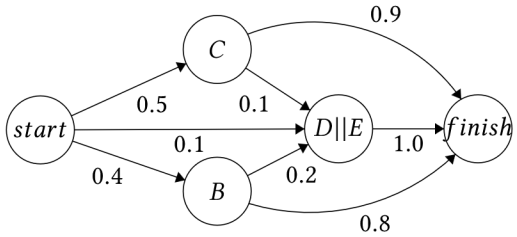



**Figure 3: Example PFA generated by Palette**

then uses Blueprint [2] to generate a full implementation of the system from these specifications.

The generated system is augmented with a GCM-based runtime that steers execution to reflect the operation of the original system. Our proposed system supports interventions by allowing users to modify these abstractions to tailor the generated system to their experimental needs, while still being representative of the original system.

## 3.1 System Topology

Palette generates a system topology from the statistics collected during trace processing; the topology encodes the calculated statistical information into an abstract representation which can be exposed to the user for further modification. It models the structural relationships within the system through a directed graph that encodes the caller-callee interactions between services and APIs, while a Probabilistic Finite Automaton (PFA) captures the diverse execution pathways of each API. The topology models the performance properties of the system by using a graphical causal model (GCM). Both structural and performance properties of the original system must be preserved to generate and execute a new representative system.

**Directed Graph Represents Topology.** The system topology is a directed graph, $G = (P, V, E)$, where $P$ is the set of all services in the system, $V$ is the set of all APIs in the system, and $E$ is the set of all edges between all APIs. Each partition in the graph, $p \in P$, encodes a single service uniquely identified by its name. Each vertex in the graph, $v \in V$, encodes a single unique API in the system and uniquely belongs to a single partition. Each vertex is uniquely identified by a combination of its name and the partition it belongs to. Each edge in the graph, $e \in E$, encodes a caller-callee relationship between any two APIs in the system. Edges between vertices in the same partition encode local function calls while edges between vertices in different partitions encode remote procedure calls.

**PFA Encodes Execution Behavior.** Palette uses a PFA to model the various execution behaviors exhibited by an API when invoking its dependencies. For instance, it can be used to distinguish between sequential and concurrent calls from one API to another; this information is not captured by the directed graph and the GCM. Figure 3 shows an example PFA generated by Palette. The PFA for each API consists of different states which the execution can be in. Every PFA has a start state that represents the start of the API execution and a finish state which represents the end of the API execution. Additionally, the PFA contains other states, each of which represents a local step in the API execution. Each state in the PFA can make one or more concurrent outgoing calls (with some probability) to dependencies. For example, in Figure 3, nodes B and C only make one call whereas node D||E make two concurrent outgoing calls. Once the calls end, the state can then transition into another state. The transition edges encode the sequential nature of the API execution. A state may transition into one of many possible

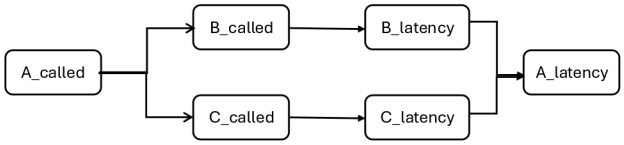



**Figure 4: Example latency causal graph**

states with a different probability. For example, in Figure 3, the start state transitions into state B with probability 0.4, state C with probability 0.5, and state D||E with probability 0.1. The PFA restricts state transitions such that the total probability of all outgoing transitions for a state sum to one. The probabilistic transition between many states encodes the ability of different execution paths.

**GCM Captures Performance Properties.** Palette includes a GCM in the topology structure of each API to encode the performance properties of the observed system. Suppose we want to model the latency of an API, A, which calls two downstream APIs, B and C with differing probabilities. Each API's latency is represented as a separate node, and the latency of A depends on the latencies of B and C, in additional to some local work. Figure 4 shows the causal graph for this simple scenario. The latency of API A is directly influenced by the latencies of B and C, so there are causal edges from their latency nodes to A's, that capture the combined impact of their latency on A's latency. Moreover, as B and C may not always be called, this is represented as 'called' nodes in the graph which represent the probability of B and C being called. Here, Palette models each 'called' node can be modeled as a simple Bernoulli distribution by default but users can override this to use more sophisticated or custom distributions. However, the causal graph does not distinguish between sequential calls to B and C from concurrent calls to B and C. This information is provided by the PFA for the API, which determines how to combine the values of the parents in the causal equation. For example, Table 2 shows the causal equation built for every latency node in the graph based on the behavior of its immediate parents. Note that different performance properties would result in different set of equations.

## 3.2 Generation mechanism

**Generating Topology From Traces.** Palette processes the trace dataset to calculate statistical information about each service from the traces including a list of all its APIs, the execution time for each API, the probability of every outgoing call for each API, and the dependencies for each service. The collected information can be further augmented with more statistics depending on the information available in traces. During the processing, Palette also builds a PFA for each API while it processes the trace data. Once the trace processing finishes, Palette coarsens the PFA by merging similar states. Palette then uses the built PFA to generate a causal graph for each API and generates a causal equation for each node in the causal graph. The causal equation is entirely dependent on the specific performance property being modeled by the GCM. Table 2 shows how the equation will be built for latency nodes, the operations might be entirely different for a different property such as payload sizes. Palette then fits a linear regression for the latencies using the observed trace data to find the coefficients (the various $\lambda_a$) used to estimate the causal effect of each parent on a given node in the graph. Our prototype uses a linear model, but users of Palette may override this by using non-linear models to represent more complex distributions by adjusting the equations in Table 2 to represent the new model.

**Topology to Specifications.** Palette then converts the topological model into actual source code that encodes the performance properties of the system. For each API, Palette converts its corresponding PFA into a set of local functions to encode PFA states and transitions. Palette also encodes a GCM model for the API and links it to the GCM-enabled runtime. For certain performance properties, Palette also encodes how that property should be achieved. For example, to achieve a sampled amount of latency, Palette chooses to perform matrix multiplications for that amount of time.

**Implementation Generation.** We combine Palette with Blueprint [2] to generate a full implementation of the system. Blueprint provides the infrastructural pieces necessary for generating a deployable version of the system.

## 3.3 Supporting Interventions

Palette supports interventions by allowing users to execute interventions at four different stages.

**Trace Processing Interventions.** Users can modify the trace processing step to filter out invalid or irrelevant edge cases. Moreover, users can augment the data dimensions processed by Palette and update the GCM generation procedure to include these dimensions. New nodes can be added to the graph to represent new dimensions of data and they can be connected to existing nodes that they influence. For instance, if an operator starts collecting data on the payload size of different API calls, and the payload size affects the latency of downstream calls, an edge between the payload size of the calling service can be added to the vertex representing the latency of the callee service. Another example is that researchers interested in a new congestion control algorithm can augment the data dimensions to include timeout values that are then reflected in the GCM and generated system, so they can understand how their algorithm behaves in a large-scale microservice.

**Topological Interventions.** The user can apply custom modifications to the topology through a set of modification primitives. For example, a researcher can add new edges and vertices to represent new causal relationships between microservices. Large topologies can also be downscaled to fit the physical testbed available to the researcher by removing edges and vertices representing the different APIs. These interventions can be useful for someone investigating how their new resource manager behaves when the call graph for a service increases in depth or width, possibly affecting the outcome of allocation decisions.

**Specification Interventions.** The user can change how a performance property is achieved. For example, to achieve desired latency, dependent on the use case, the user might want its services to sleep and not do any work or they might want the services to add work that is memory bound.

**Instantiation Interventions.** Users can add further interventions to the system by modifying the Blueprint IR to modify concrete service implementations. For example, the IR can be modified to support different RPC frameworks that change how the actual benchmark system gets instantiated.

### 3.4 GCM-Based Runtime

Each API leverages the causal equation and coefficients produced by Palette to drive system execution in a way that resembles the original system behavior. For example, the runtime samples the GCM at each API to obtain values such as the execution time or the payload size for outgoing messages. These samples are conditioned on the causal relationships encoded in the GCM, allowing the system to better capture dependencies than sampling from the observed distributions.

**Live Measurements.** Palette makes measurements for different nodes of the causal graph during the execution and uses these measurements into the local causal equations at every API to make predictions about the expected behavior. For example, consider the latency scenario from Figure 4. In the implementation of API A, Palette will insert code to measure the latency of the outgoing calls to B and C. Palette will then plug the actual measured/observed values into the causal equation to find the expected latency of A.

**Causal Baggage Propagation.** The causal graphs of some APIs might require measurements made higher up in the call chain that are not directly measurable at the service executing the specific API. To ensure that all the relevant information is correctly propagated by upstream nodes to the downstream nodes, Palette uses baggage propagation [25] to propagate the required measurements to the downstream nodes executing the API. Palette can compile and inject the exact code for adding the causal data into the baggage and propagate it downstream along with the request because the dependency graph is known at generation time.

**Live Corrections.** During the system execution, systems may deviate from the expected behavior of the system. For example, a timeout may be triggered in one of the APIs due to different performance characteristics of the deployed hardware. GCMs enable live corrections during runtime when execution diverges from the behavior observed in the original traces. The model enables more informed sampling decisions about system behavior by conditioning the sampling process on the underlying causal relationships that influence the generated metrics.

## 4 Conclusions

In this paper, we have presented the design of an extensible system for generating representative macrobenchmark microservice systems from distributed trace datasets that uses Graphical Causal Models (GCMs) for modeling system behavior to preserve the desired representative characteristics of a microservice systems. We believe that Palette provides a flexible way for users to conduct intervention experiments to validate and test new advancements for microservices.

## Acknowledgments

We would like to thank the anonymous reviewers for their invaluable feedback and for helping us shape the final version of the paper.

## References

[1] Abdullah Alomar, Pouya Hamadanian, Arash Nasr-Esfahany, Anish Agarwal, Mohammad Alizadeh, and Devavrat Shah. {CausalSim}: A causal framework for unbiased {Trace-Driven} simulation. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1115–1147, 2023.

[2] Vaastav Anand, Deepak Garg, Antoine Kaufmann, and Jonathan Mace. Blueprint: A toolchain for highly-reconfigurable microservice applications. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 482–497, 2023.

[3] Kailash Budhathoki, Dominik Janzing, Patrick Bloebaum, and Hoiyi Ng. Why did the distribution change? In *International Conference on Artificial Intelligence and Statistics*, pages 1666–1674. PMLR, 2021.

[4] Kailash Budhathoki, Lenon Minorics, Patrick Blöbaum, and Dominik Janzing. Causal structure-based root cause analysis of outliers. In *International conference on machine learning*, pages 2357–2369. PMLR, 2022.

[5] Adrian Cockcroft. The evolution of microservices. (April 2016). Retrieved October 2020 from https://www.slideshare.net/adriancockcroft/evolution-of-microservices-craft-conference, 2016.

[6] Adrian Cockcroft. Microservices workshop: Why, what, and how to get there. (April 2016). Retrieved October 2020 from https://www.slideshare.net/adriancockcroft/microservices-workshop-craft-conference, 2016.

[7] Andrea Detti, Ludovico Funari, and Luca Petrucci. $\mu$bench: An open-source factory of benchmark microservice applications. *IEEE Transactions on Parallel and Distributed Systems*, 34(3):968–980, 2023.

[8] DoWhy documentation v0.8. Root cause analysis (rca) of latencies in a microservice architecture. https://www.pywhy.org/dowhy/v0.8/

example_notebooks/rca_microservice_architecture.html, 2024. [Accessed 05-06-2025].

[9] Fanrong Du, Jiuchen Shi, Quan Chen, Li Li, and Minyi Guo. A microservice graph generator with production characteristics, 2024.

[10] Felix Elwert. Graphical causal models. In *Handbook of causal analysis for social research*, pages 245–273. Springer, 2013.

[11] Rodrigo Fonseca, George Porter, Randy H Katz, and Scott Shenker. {X-Trace}: A pervasive network tracing framework. In *4th USENIX Symposium on Networked Systems Design & Implementation (NSDI 07)*, 2007.

[12] Joshua Fried, Gohar Irfan Chaudhry, Enrique Saurez, Esha Choukse, Íñigo Goiri, Sameh Elnikety, Rodrigo Fonseca, and Adam Belay. Making kernel bypass practical for the cloud with junction. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 55–73, 2024.

[13] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 3–18, 2019.

[14] Einas Haddad. Service-oriented architecture: Scaling the uber engineering codebase as we grow. (September 2015). Retrieved October 2020 from https://eng.uber.com/service-oriented-architecture/, 2015.

[15] Mazdak Hashemi. (January 2017). Retrieved February 2021 from https://blog.twitter.com/engineering/en_us/topics/infrastructure/2017/the-infrastructure-behind-twitter-scale.html, 2017.

[16] Darby Huye, Yuri Shkuro, and Raja R Sambasivan. Lifting the veil on {Meta's} microservice architecture: Analyses of topology and request workflows. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 419–432, 2023.

[17] Dominik Janzing, Kailash Budhathoki, Lenon Minorics, and Patrick Blöbaum. Causal structure based root cause analysis of outliers. *arXiv preprint arXiv:1912.02724*, 2019.

[18] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter RPCs can be general and fast. In *16th USENIX Symposium on Networked Systems Design and Implementation*, NSDI, 2019.

[19] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr. Sharma, Arvind Krishnamurthy, and Thomas Anderson. TAS: TCP acceleration as an OS service. In *14th ACM European Conference on Computer Systems*, EuroSys, 2019.

[20] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. R2P2: Making RPCs first-class datacenter citizens. In *2019 USENIX Annual Technical Conference*, ATC, 2019.

[21] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan MG Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, et al. Swift: Delay is simple and effective for congestion control in the datacenter. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 514–528, 2020.

[22] Pedro Las-Casas, Giorgi Papakerashvili, Vaastav Anand, and Jonathan Mace. Sifter: Scalable sampling for distributed traces, without feature engineering. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 312–324, 2019.

[23] I-Ting Angelina Lee, Zhizhou Zhang, Abhishek Parwal, and Milind Chabbi. The tale of errors in microservices. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 8(3):1–36, 2024.

[24] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. Characterizing microservice dependency and performance: Alibaba trace analysis. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 412–426, 2021.

[25] Jonathan Mace and Rodrigo Fonseca. Universal context propagation for distributed system instrumentation. In *Proceedings of the thirteenth EuroSys conference*, pages 1–18, 2018.

[26] microservices demo. Sockshop. Retrieved August 2022 from https://github.com/microservices-demo/microservices-demo, 2016.

[27] Haoran Qiu, Subho S Banerjee, Saurabh Jha, Zbigniew T Kalbarczyk, and Ravishankar K Iyer. {FIRM}: An intelligent fine-grained resource management framework for {SLO-Oriented} microservices. In *14th USENIX symposium on operating systems design and implementation (OSDI 20)*, pages 805–825, 2020.

[28] Sultan Mahmud Sajal, Timothy Zhu, Bhuvan Urgaonkar, and Siddhartha Sen. Traceupscaler: Upscaling traces to evaluate systems at high load. In *Proceedings of the Nineteenth European Conference on Computer Systems*, pages 942–961, 2024.

[29] Korakit Seemakhupt, Brent E Stephens, Samira Khan, Sihang Liu, Hassan Wassel, Soheil Hassas Yeganeh, Alex C Snoeren, Arvind Krishnamurthy, David E Culler, and Henry M Levy. A cloud-scale characterization of remote procedure calls. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 498–514, 2023.

[30] Vishwanath Seshagiri, Darby Huye, Lan Liu, Avani Wildani, and Raja R Sambasivan. [sok] identifying mismatches between microservice testbeds and industrial perceptions of microservices. *Journal of Systems Research*, 2(1), 2022.

[31] Yuri Shkuro. Microsim - microservices simulator. Accessed June 2025 from https://github.com/yurishkuro/microsim, 2018.

[32] Gagan Somashekar, Karan Tandon, Anush Kini, Chieh-Chun Chang, Petr Husak, Ranjita Bhagwan, Mayukh Das, Anshul Gandhi, and Nagarajan Natarajan. {OPPerTune}:{Post-Deployment} configuration tuning of services made easy. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 1101–1120, 2024.

[33] Akshitha Sriraman, Abhishek Dhanotia, and Thomas F Wenisch. Softsku: Optimizing server architectures for microservice diversity@ scale. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 513–526, 2019.

[34] Akshitha Sriraman and Thomas F Wenisch. {$\mu$Tune}:{Auto-Tuned} threading for {OLDI} microservices. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 177–194, 2018.

[35] Mert Toslali, Emre Ates, Alex Ellis, Zhaoqi Zhang, Darby Huye, Lan Liu, Samantha Puterman, Ayse K Coskun, and Raja R Sambasivan. Automating instrumentation choices for performance problems in distributed applications with vaif. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 61–75, 2021.

[36] Zhiqiang Xie, Yujia Zheng, Lizi Ottens, Kun Zhang, Christos Kozyrakis, and Jonathan Mace. Cloud atlas: Efficient fault localization for cloud systems using language models and causal insight. *arXiv preprint arXiv:2407.08694*, 2024.

[37] Haoran Zhang, Konstantinos Kallas, Spyros Pavlatos, Rajeev Alur, Sebastian Angel, and Vincent Liu. {MuCache}: A general framework for caching in microservice graphs. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 221–238, 2024.

[38] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, et al. The demikernel datapath os architecture for microsecond-scale datacenter systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 195–211, 2021.

[39] Lei Zhang, Zhiqiang Xie, Vaastav Anand, Ymir Vigfusson, and Jonathan Mace. The benefit of hindsight: Tracing {Edge-Cases} in distributed

systems. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 321–339, 2023.

[40] Yazhuo Zhang, Rebecca Isaacs, Yao Yue, Juncheng Yang, Lei Zhang, and Ymir Vigfusson. Latenseer: Causal modeling of end-to-end latency distributions by harnessing distributed tracing. In *Proceedings of the 2023 ACM Symposium on Cloud Computing*, pages 502–519, 2023.

[41] Zhizhou Zhang, Murali Krishna Ramanathan, Prithvi Raj, Abhishek Parwal, Timothy Sherwood, and Milind Chabbi. {CRISP}: Critical path analysis of {Large-Scale} microservice architectures. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 655–672, 2022.

[42] Ziming Zhao, Zhenwei Wang, Tiehua Zhang, Zhishu Shen, Hai Dong, Zhen Lei, Xingjun Ma, Gaowei Xu, Zhijun Ding, and Yun Yang. Chase: A causal hypergraph based framework for root cause analysis in multimodal microservice systems, 2025.

[43] Renjie Zhou, Dezun Dong, Shan Huang, and Yang Bai. Fasttune: Timely and precise congestion control in data center network. In *2021 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)*, pages 238–245. IEEE, 2021.

[44] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chenjie Xu, Chao Ji, and Wenyun Zhao. Poster: Benchmarking microservice systems for software engineering research. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, pages 323–324. IEEE, 2018.