Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master Thesis

Mewbie: Scale Adjustable Benchmark for Microservice Deployments

Author: Ritul Satish (2801882)

1st reader:	Prof. dr. Alexandru Iosup
2nd reader:	Dr. Daniele Bonetta
Daily supervisor:	Sacheendra Talluri, MSc
External Supervisors:	Prof. dr. Rodrigo Miragaia Rodrigues
	Dr. João Garcia
	João Loff, MSc (INESC-ID, Lisbon)

A thesis submitted in fulfillment of the requirements for the joint UvA-VU Master of Science degree in Computer Science

October 14, 2025

"He who can listen to the music in the midst of noise can achieve great things."
by Dr. Vikram Sarabhai

Abstract

Microservice architectures have become essential for modern software systems and services across diverse sectors. While they offer advantages in scalability and agility, their inherent complexity and massive scale make it hard to understand the performance implications of system architectural and design choices. This limitation affects researchers and developers even more severely, as they lack access to realistic production environments.

Furthermore, while a few microservice benchmarks exist, they fall short of this goal as they are based on simplified applications, orders of magnitude smaller than real-world deployments.

We present Mewbie, the first scale-adjustable benchmark that leverages production traces to create flexible and representative microservice workloads. Mewbie is based on very large scale production traces, but scales to any deployment size through a novel trace downscaling technique that preserves key topological features of the original trace, enabling realistic benchmarking of the various components of a microservice architecture without the overhead of a full-scale deployment.

We showcase Mewbie in a datastore-centric scenario, exploring how a microservice deployment based on Alibaba's production dataset responds to varying datastore mixtures (Redis, MongoDB, and MySQL) and diverse consistency semantics. Our evaluation focuses on Mewbie's ability to simplify large traces, measure the effects of different system designs, and adapt to various service configurations. Together, these capabilities demonstrate how Mewbie helps uncover the performance impact of architectural decisions.

Contents

1

Introduction

Microservice architectures were initially adopted by large-scale service providers, such as Meta (1), Uber (2), Netflix (3), and Alibaba (4), and have since evolved into the *de facto* standard for building modern web-based applications. Microservices deliver several benefits such as flexibility, resilience, and independent development, deployment, and scalability – qualities that have driven their widespread adoption across diverse use cases and deployment sizes, from niche applications to enterprise-level systems (5). However, these same advantages can result in inherently complex deployments, as multiple services must coordinate, communicate, and evolve organically.

One major advantage of microservices is that they allow independent scalability (6). Each service runs independently with resources required for its smooth performance. Developers can scale only the parts of the systems that need more resources. For example, netflix can scale its video streaming service during peak hours without having to scale other services like payment or recommendation engine in a similar way (7). This helps avoid over-provisioning of resources while still delivering smooth performance to the users (8).

Another key advantage of microservices is fault isolation (6). If one service fails, the rest of the system is functionally thus avoiding full system outages in the case of service crashes. Amazon uses this approach, this way, if the payment service has an issue, users can still browse and add products to their carts (9).

These advantages, though powerful, come at the high cost of increased architectural complexity, which in turn demands robust strategies for monitoring, testing, and performance evaluation.

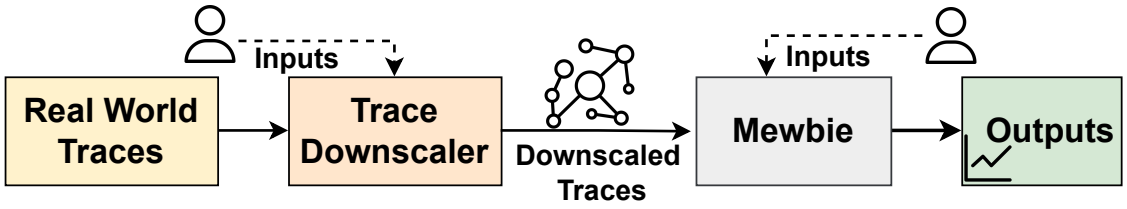

Figure 1.1: High-level Mewbie benchmarking pipeline.

1.1 Problem Statement

Identifying best practices and evaluating trade-offs between scalability, performance and correctness in different design alternatives remains a significant challenge in microservice architectures (10, 11, 12). These challenges stem from coordinating multiple services that may use distinct communication protocols, datastores, and frameworks – all of which increase the complexity of diagnosis and optimization. On top of that, microservices in real-world deployments often reach an impressive scale (2, 4), far beyond that of traditional distributed systems. This makes it hard to identify potential issues early and to validate whether proposed solutions will hold up at production scale, thus hindering design iteration and deployment speed.

which makes it difficult not only to discover potential problems ahead of time, but also to ensure that proposed solutions will actually work at production-level size.

Given the growing importance of microservice architectures, several benchmarking efforts have emerged – aiming to evaluate performance, resource utilization, and overall behavior under realistic conditions. Such benchmarks are of great importance, as they enable identifying performance bottlenecks, comparing architectural approaches, and highlighting optimal configurations – much like what YCSB (13) did for the datastore community. However, benchmarking microservices is inherently more complicated than benchmarking databases. In particular, simply increasing the number of client requests or object size does not suffice to evaluate scalability. Microservices involve heterogeneous datastores (e.g., caching vs. relational databases), complex communication patterns (e.g., synchronous vs. asynchronous), and various service roles (stateless vs. stateful).

To address these issues, a few microservice benchmarks emerged, including DeathStarBench (14), μSuite (15), TeaStore (16), and Acme Air (17). While these benchmarks have the nice feature of being inspired by realistic applications such as e-commerce or social networks, they all fail to capture *true* real-world scale and properties (as summarized

Table 1.1: Comparison of existing demo and benchmark applications with Mewbie.

Application	Category	# Services	Workload	Customization Effort
Industry				
Alibaba (4)	E-Commerce	∼17'000	Real	–
Uber (2, 18)	Transport Service	∼4'500	Real	–
Netflix (3)	Media Streaming	∼700	Real	–
Ebay (19)	Online Auctioning	∼1'000	Real	–
Monzo (20)	Banking	∼1'500	Real	–
Demo				
Online Boutique (21)	E-Commerce	∼10	None	High
eShop (22)	E-Commerce	∼10	None	High
BookInfo (23)	Catalog	<10	None	High
HotRod (24)	Transport Service	<10	None	High
Benchmarks				
Acme Air (25)	Reservation Service	<10	Artificial	High
μSuite (15)	Data Processing	10-50	Artificial	High
DeathStarBench (14)	Social Network Media Services Hotel Reservation	10-40	Artificial	High
TeaStore (16)	E-Commerce	<10	Artificial	High
TrainTicket (26)	Reservation Service	∼50	Artificial	High
Mewbie	**Any**	**Custom**	**Based on Real**	**Low-Medium**

in table 1.1). For example, most applications of DeathStarBench use less than 25 services each, which starkly contrasts with over 17,000 services found in Alibaba's production traces (4) or the 4,000+ services at Uber (2).

This leaves developers and researchers stuck in a binary choice of testing their architecture and code at an unrealistic scale using those benchmarks – which cannot reliably predict system behavior under realistic loads – or invest significantly and without much guidance in approximating a production environment at the scale of their specific testing needs. Furthermore, the actual target scale can vary significantly, based on the resources available to developers, which might be limited by factors such as cloud infrastructure costs. As such, having the flexibility to scale up or down within a wide range is desirable, as it would allow developers to use their available resources to approximate the characteristics of real-world microservice deployments, even if they cannot replicate every nuance of massive production environments.

We identify four core problems:

- Existing benchmarks do not reflect the scale and operational complexity of real-world microservice deployments.

- Simulating production environments is expensive, both in terms of infrastructure costs and developer efforts.
- There is a lack of customizable benchmarking frameworks that adapt to varying resource constraints.
- Open source microservice traces often lack critical information such as request semantics, operation details or service roles, thus hindering their use for realistic and representative benchmarking.

1.2 Research Questions

The goal of this work is to answer the **Main Research Question (MRQ):** *How can we design a scale-adjustable and customizable microservice benchmarking framework that realistically captures real world application characteristics?*

Given the challenges outlined in 1.1, the objective of this work is to propose, implement, and evaluate a novel benchmarking system, **Mewbie**, that leverages real-world traces, supports variable scale, and enables meaningful performance analysis of microservice-based systems. This main goal is addressed through three subquestions, each aligned with a specific problem statement (P1–P4):

This MRQ is addressed through the following subquestions:

1. **Research Question 1 (RQ1):** *How to generate realistic and usable benchmarks and workloads from production-scale microservice traces?* As identified in **P4**, publicly available microservice traces often lack essential information required to simulate application behavior, such as object identifiers, operation semantics, and service roles. This question focuses on the development of a trace enrichment tool that suplements existing traces with such critical information. This enables the generation of a realistic and representative benchmark application and workloads.
2. **Research Question 2 (RQ2):** *How to design and implement a benchmarking framework that enables scale-adjustability, customizablity, and automated deployment?* **P2** highlights the challenges of replicating production-scale environments, while **P3** emphasizes the lack of customizable benchmarking tools that adapt to different infrastructure constraints and target benchmarking scenarios. This research question aims to answer the design and implementation of a modular, reusable benchmarking system that allows users to define custom workloads, adjust architectural features, and deploy them with minimal manual effort.

3. **Research Question 3 (RQ3):** *How effective is the Mewbie benchmark in uncovering architectural tradeoffs and capturing the performance impact of service-level configuration changes in microservice-based systems?* As identified in **P1**, the lack of realistic, scale-adjustable benchmarks for microservice systems, this question investigates Mewbie's effectiveness as a performance analysis tool. By leveraging enriched and downscaled real-world traces, we examine how well Mewbie reveals architectural tradeoffs, uncovers service-level bottlenecks, and captures the impact of configuration changes.

1.3 Research Methodology

System Design & Implementation: To address **RQ1** and **RQ2**, we adopt an iterative and structured design approach, guided by the AtLarge Design Process (27), in combination with established software engineering practices as described by Sommerville (28) and Bass et al. (29). The process consists of: (i) identifying design challenges , (ii) designing architectural components and interfaces, (iii) implementing prototypes of key components, and (iv) iterating the design and validation process until the research questions are answered.

Experimental Research: To address **RQ3**, we adopt an experimental research methodology to evaluate the effectiveness of Mewbie in realistic scenarios. This includes: (i) defining relevant performance metrics such as request completion time, and application throughput (ii) generate representative benchmarks and workloads, (iii) deploying the benchmark infrastructure developed in **RQ2** across varying configurations and scales, (iv) conducting experiments that modify architectural features and service-level settings (e.g., communication mode, consistency level), and finally (v) analyzing the resulting performance data to identify tradeoffs and bottlenecks.

Open Science: The Mewbie framework is released as open-source software[1]. This commitment to open science ensures transparency, supports reproducibility, and encourages future work that builds upon or extends our system. Evaluation data and configuration details are also made available to enable independent validation and comparison.

[1] `https://github.com/ritulS/MS_Benchmark`

1.4 Thesis Contributions

To address problems raised (P1-P4) and to answer the research questions (RQ1-RQ3), we propose Mewbie, a benchmark for microservice ecosystems, which is the first benchmark designed to capture the scale and complexity of real-world microservice deployments. When developing Mewbie, instead of taking the conventional application-centric approach, which is hard to scale to realistic deployment sizes, we propose a novel trace-driven design, which leverages real-world production traces to create a benchmark based on modifiable skeleton applications on top of a custom-scaled deployment.

More precisely, our benchmark generation method begins with existing traces from large-scale Internet service providers – such as those released by Alibaba (4) – which capture real user interactions. This approach enables us to replicate authentic request flows across services and accurately mirror realistic service-to-service communication patterns. Then, to create a practical benchmark, we addressed several technical challenges.

First, these traces often lack some of the required information to build a complete benchmark, such as the application logic, underlying platforms, object identifiers, etc. To overcome this, we developed a trace enrichment tool that enhances existing traces by supplementing them with the information needed for a comprehensive analysis.

Second, the scale of the services operated by the above-mentioned real-world providers can be overwhelming for duplication by a benchmark, in terms both of the time it might take to run the benchmark, and, more importantly, of the monetary cost for purchasing or accessing the respective resources. As such, we moved away from a fixed-scale benchmark, opting instead for a flexible approach where the users of the benchmark can configure its size according to their needs.

Finally, this flexibility introduces the challenge of downscaling the original traces that Mewbie relies on, while still preserving their essential characteristics. In this respect, traditional graph downscaling strategies like aggregation and coarsening tend to combine multiple nodes into a single node (30, 31, 32), obscuring their individual characteristics and losing information about request flow and contention. To address this, we developed a downscaling technique tailored for microservice traces, with the goal of enabling users to work with reduced, yet representative, versions of real-world traces.

We demonstrate the practical benefits of Mewbie through experiments using a down-scaled Alibaba trace (4) and experimenting with different architectural decisions, such as datastore and communication patterns, or different service-level configurations such as datastore consistency settings. Our experimental evaluation shows how Mewbie is able to

showcase the performance tradeoffs of these decisions, and also to unveil covert bottlenecks and cascading effects.

In summary, this paper makes the following contributions:

- We present Mewbie, the first benchmark for microservices that attempts to be representative of the scale and complexity of real-world deployments. The design of Mewbie uses production grade traces and supplements them with information like object identifiers, stateful operations, read-write ratio, service types and async-sync ratio. Mewbie is available as open source and can be instantiated in a similar manner to popular benchmarks such as YCSB (13).
- We introduce a novel trace downscaling approach that reduces production microservice traces to more manageable and deployable sizes, while preserving essential topological features of the call graph. Our method combines stratified and priority sampling to select a representative subset of trace data. As shown in our evaluation, it outperforms simpler sampling approaches by maintaining essential structural metrics – such as trace depth, degree distribution, and clustering coefficient – relative to the original graph.
- We streamline the generation, deployment, and configuration processes using our trace enrichment and skeleton application. Mewbie effortlessly incorporates deployment-level information into existing traces, enabling rapid benchmark creation tailored to the user's specific goals. Our experimental results shows that Mewbie is able to highlight performance differences at the architectural and service level.

1.5 Societal Relevance

Answering the **MRQ** is not only of technical importance but also directly supports several of the grand societal challenges outlined in the CompSys NL Manifesto (33). This work directly addresses three out of the four challenges presented in the manifesto:- **Responsibility**, **Sustainability** and **Usability**.

This work directly contributes toward **Challenge 2: Responsibility**. Microservice systems power critical infrastrucure like transportation, healthcare, banking and public services, therefore their availability, performance and resilience is essential for societal well-being. By enabling realistic performance evaluation through representative benchmarks, this research supports the design of more dependable and responsible computing infrastructures.

The benchmark's scale-adjustability contributes to **Challenge 3: Sustainability**. In large scale microservice application, inefficient designs and misconfigured services often result in wasted computational and hence excess energy consumption. This research develops a benchmarking framework that allows for fine-grained iterative evaluation of different architectural and configuration scenarios, allowing organizations to tune systems for better energy efficiency.

Finally, the customizable nature of the benchmark aligns with **Challenge 4: Usability**. By supporting performance evaluation at varying deployment scales, the benchmark becomes accessible not only to large enterprises but also to smaller research labs and startups.

Overall, this research supports the society in several ways by enabling the reliability, efficiency, and accessibility of large scale microservice systems that make up our digital world.

1.6 Plagiarism Declaration

I hereby declare that this thesis is the result of my own independent work and writing. Unless explicitly cited, it does not include content copied from any external sources. Furthermore, this work has not been submitted for evaluation in any other context.

1.7 Thesis Structure

The thesis is organized as follows. In chapter 2, we introduce the high-level design of Mewbie. Next, chapter 3 details our trace downscaling process, which is followed by a discussion of our benchmark generation approach in section 4.1, covering key design aspects of both our trace enrichment and skeleton application components. We then present the benchmark deployment and usage in section 4.2 and evaluate Mewbie's downscaling and configuration processes in chapter 5. Finally, section 6.2 concludes the paper with a discussion and final remarks.

2

Mewbie Design

Mewbie is a benchmarking framework that enables realistic testing and evaluation of microservice architectures at scale. We begin by outlining the main requirements behind Mewbie (section 2.1), followed by an overview of its architecture (section 2.2) and finally detail the key metrics Mewbie logs and stores (section 2.3).

2.1 Requirements

To better guide the design of the Mewbie benchmark framework, we outline the following set of requirements.

(R1) Scale Adjustability: System performance often degrades unpredictably at scale due to resource contention, service fanout, and network delays (34, 35). Microservice interactions that appear efficient at small scale can result in bottlenecks or instability as the system grows. However, as mentioned, evaluating system behavior at production scale is often infeasible in research or software development environments due to infrastructure limitations (4). To address this, Mewbie must support scale-adjustable benchmarking, allowing users to simulate systems of varying size from small testbeds to large-scale deployments based on available compute resources. This can provide insights to uncover scale sensitive issues such as tail-latency spikes, database overload, or cascading failures without the associated costs and risks of testing in production (36).

(R2) Customizability: Microservice systems involve a wide and evolving set of design choices including databases, caches, communication protocols, replication strategies, and consistency models (37). Evaluating the trade-offs of these design choices not only requires the ability to configure multiple system parameters, but also the flexibility to introduce or replace system components as technologies evolve (38). To support this, Mewbie must

allow users to customize the benchmark at multiple levels: trace size, service settings (e.g., consistency model), call-level semantics (e.g., sync or async), and deployment level parameters like (e.g., CPU limits). Additionally Mewbie must support a modular architecture where individual components such as stateless services, databases and caches can be independently extended or swapped.

(R3) Application-agnosticism: Existing microservice benchmarks are often restricted to specific domains such as e-commerce or social networks (14, 15, 16, 17), which raises the question of their applicability across broader use cases. This domain specificity limits researchers and developers from evaluating systems in contexts that differ operationally; for example applications in scientific computing or healthcare, which often involve different communication patterns or service dependencies (39). To address this, Mewbie was designed to be application-agnostic, so it can it be configured to approximate different production workloads.

(R4) Readiness: Existing benchmarks often require significant manual setup before they can be used effectively. Mewbie should overcome this by providing built-in tools for workload generation, deployment, telemetry, making it easy to evaluate system designs with minimal effort.

2.2 Architecture Overview

fig. 2.1 illustrates the architecture of the Mewbie framework. Its workflow is organized into two distinct phases: benchmark generation and benchmark execution. In the benchmark generation phase, the user-provided traces and customization options are used to generate a benchmark. This generated benchmark is then executed in the execution phase (as many times as desired to evaluate a system design).

Benchmark Generation. In the generation phase, Mewbie takes as input a microservice trace (as detailed in chapter 3) along with a set of user-defined benchmark configuration options that specify the desired system behavior and deployment constraints. Raw traces are often infeasible for benchmarking due to their large scale and missing execution semantics, such as service types, operation and payload details. Mewbie addresses these challenges using two key tools: trace downscaling and trace enrichment.

The trace downscaling tool ① (see fig. 2.1) reduces the size of the trace graph to a manageable scale based on the user's available computational resources. It uses graph sampling techniques to generate a smaller, deployable version of the original trace that retains essential structural characteristics such as service fanout, depth, and interaction

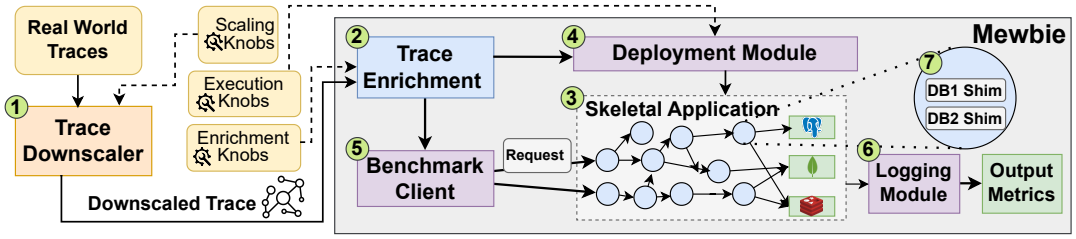

Figure 2.1: Mewbie architecture.

patterns. This enables realistic benchmarking at a fraction of the original scale, satisfying **R1**. We further detail this technique in chapter 3.

Mewbie's Trace Enrichment tool ② enhances the downscaled trace by filling in execution level details based on user input. These details include service types, operation type and data payloads. From this enriched trace, the tool generates enriched request call graphs. An enriched request call graph captures the full execution plan of a client-initiated request, including the sequence of services to be invoked, the type of operation at each hop (e.g., read vs. write, or synchronous vs. asynchronous), and any other required metadata (e.g., payload size). The set of all enriched request call graphs forms the benchmark workload. Importantly, this enrichment process is customizable. Users can also define how services are mapped (e.g., databases, stateless services), specify operation semantics and control workload properties such as object sizes and access patterns. This customizability enables targeted benchmarking of different architectural choices, satisfying **R2**.

Benchmark Execution. Once the benchmark is generated, Mewbie moves to the execution phase, where the skeletal application ③ and workload are deployed and run. The skeletal application is a containerized placeholder code that emulates the original service while preserving its interaction patterns with other related services, thus satisfying **R3**.

The skeletal application is deployed using Mewbie's Deployment Module ④, which automatically deploys and configures the service nodes based on user-defined deployment settings such as CPU/memory limits. Furthermore, Mewbie includes a benchmark client ⑤ that injects the request into the skeletal application, to create the end-to-end request workflows. Throughout the execution, Mewbie's Logging Module ⑥ captures detailed performance telemetry, including metrics such as request completion time, service-level latencies and system throughput. Finally, Mewbie includes a set of Shim Components ⑦, which are wrappers that define the interaction logic with various service types. These shims manage connection setup, query execution and response handling with service-specific code. As

such, by customizing or extending these shims, users can easily incorporate new services or modify the existing ones.

By automating deployment, workload generation, and metric collection, the Mewbie execution pipeline enables repeatable evaluation of microservice systems with little effort. This makes the benchmarks easily usable, satisfying **R4**.

2.3 Mewbie Target Metrics

Request Completion Time. In microservice architectures, a single user request often triggers a complex chain of service calls across the system (2). Prior studies show that latency beyond acceptable thresholds can significantly degrade user satisfaction (40, 41, 42). This makes request completion time a fundamental performance metric, directly impacting system responsiveness and user experience.

Mewbie records timestamps at the start of a request and at each service node during its propagation. The completion time is computed as the difference between the client-issued timestamp and the maximum timestamp at the leaf nodes in the request's call graph. High completion times may signal queueing delays, resource contention, or other inefficiencies.

Application Throughput. Throughput measures the rate at which the system completes end-to-end requests. Unlike monolithic systems, where throughput impacts are relatively easy to identify, microservice architectures involve complex chains of service interactions, including synchronous and asynchronous calls, retries and coordination mechanisms (43). This complexity makes it difficult to pinpoint bottlenecks, as reduced throughput might emerge from queueing, resource contention or cascading slowdowns/failures at various points in the system (44, 45, 46).

Mewbie helps address this complexity by enabling controlled experimentation on these behaviors, allowing developers to test how design changes such as switching datastores or modifying consistency levels impact throughput – which is often hard to determine in production settings (47).

3

Trace Downscaling

The design of the trace downscaling method was submitted to satisfy the requirements for 'Industrial Internship' (6 EC)

Microservice traces are often large and complex, containing thousands of services and millions of unique nodes, each representing a distinct service invocation. For example, a single 8-hour trace from Alibaba's production system includes over 17,000 services (4). To make benchmarking feasible across a broader range of environments than the original large-scale infrastructure, we design and implement a trace downscaler. The goal of this stage is to reduce the size of a raw microservice trace while preserving the characteristics of the original. This addresses Research Question 1 (RQ1), and this section presents our approach to solving it.

Trace definition. A microservice trace captures the set of end-to-end requests that flow through a microservice system. Each request that arrives in the entire ecosystem triggers a request call graph, which is a directed acyclic graph (DAG), where nodes correspond to services and edges denote inter-service RPC/REST calls (see fig. 3.1). Then, we define a trace as the union of all such request call graphs that are collected over a period of time from production systems, typically done using distributed tracing infrastructure such as OpenTelemetry (48).

3.1 Limitations of Existing Techniques.

Traditional graph reduction techniques, such as aggregation and coarsening (30, 31, 32), are ill-suited for microservice trace graphs due to the unique semantic and structural constraints of such data. In this context, nodes represent microservice instances (e.g.,

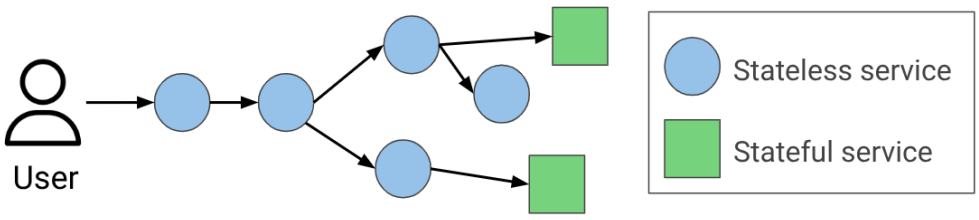

Figure 3.1: Request Call Graph Structure.

front-end, database, caching layers) and edges represent directional service-to-service associations, typically API calls that include ordering and other relevant metadata.

The limitation of these reduction methods is that they operate by merging nodes or simplifying edge structures to downscale the graph. For instance, aggregation may group multiple microservice nodes into a super-node based on structural similarity or proximity in the topology. However, this process can break the association between nodes and individual requests, making it impossible to determine which microservices participated in a given request — a critical requirement for trace analysis in microservice architectures, where request-level visibility is essential. Similarly, coarsening techniques drop or relax edge directionality, thereby losing caller–callee relationships that encode the temporal order of service execution within a trace.

Additionally, microservice traces often exhibit functional diversity, where different types of services (e.g. queues, storage) appear within a request call graph. Reducing the trace (graph) without regard for this diversity can flatten distinctions between service roles, which are often crucial for understanding latency bottlenecks or system failures.

3.2 Topological Properties of Microservice Traces.

Recent measurement studies highlighted several relevant characteristics of microservice traces, such as heterogeneous fan-in and fan-out patterns, deep invocation chains, and localized clusters of interdependent services (2). These structural patterns often reflect design choices like service co-location, or shared caching and storage layers, which can significantly influence system behavior under load. For a downscaled trace to serve as a meaningful substitute in a microservice benchmark, we would like to retain the structural properties of the original trace, not just for realism, but also to ensure that behaviors such as queue buildup, retry storms, or cross-service contention are faithfully preserved

in simulations. To approach this goal in a systematic way, we highlight several graph properties that characterize a trace of calls across services.

Call depth represents the number of hops a request takes, revealing the complexity of the request. High call depth often correlates with increased tail latency and higher sensitivity to stragglers (4). *Degree distributions* measure how many services each request flowing through a service makes calls to, capturing communication intensity. Skewed degree distributions can reveal services that handle too much traffic or act as central points of coordination, which goes against the idea that microservices should be independent and loosely coupled (2, 49). The *in-degree* of the request graph highlights the hotness of service nodes, while its *out-degree* captures service branching and downstream dependencies. Finally, the *clustering coefficient* of the graph reflects the tendency of services to form tightly connected groups. Mewbie's downscaling approach is designed to preserve these properties, ensuring that the scaled down version reflects the request patterns of the original trace. By doing this, we ensure that the downscaled trace remains faithful to the original system's architectural topology, thus yielding more accurate insights into performance bottlenecks and other behaviour.

3.3 Our approach: Microservice Trace focused Downscaling

To overcome the limitations of traditional methods while preserving the aforementioned topological properties, we design a custom graph downscaling technique tailored to the unique structure of microservice traces. Our approach provides user control over the target trace size, allowing specification of the desired number of services (n), number of requests (t'), and service composition (e.g., the ratio of stateful to stateless services).

Inspired by the graph sampling techniques of Leskovec et al. (50), our approach uses a three-phase downscaling pipeline comprising scoring, sampling, and pruning. In the scoring phase, importance values are assigned to service nodes and request call graphs. The sampling phase then selects a representative subset (subgraph) guided by these scores and user-defined constraints. Finally, the pruning phase removes unselected nodes and edges, yielding a simplified yet structurally faithful downscaled trace. fig. 3.2 shows the execution flow of the downscaling processs.

3.3.1 Scoring Phase

To guide the downscaling process, we first assess the relative importance of nodes and request call graphs in a microservice trace. Scoring both is necessary because the goal

Table 3.1: Topological properties and their significance (50).

Topological Property	Significance
Call Depth Distribution	Hotness/Coldness of nodes
In-degree Distribution	Fan-in patterns
Out-degree Distribution	Fan-out patterns
Clustering Coefficient	Connectedness of Services

is to preserve not only individual services that play critical functions, but also representative end-to-end request paths that reflect typical execution behavior. Therefore, the scoring process proceeds in two steps: computing the Node Importance Score (NIS) and the Request Importance Score (RIS).

The **Node Importance Score (NIS)** quantifies the relevance of a node based on both its frequency of use and its role in the system. The primary component of NIS is a popularity score, defined as the number of unique request call graphs in which the node appears. This captures the intuition that nodes invoked across many requests are structurally important to system behavior. Furthermore, to account for user preferences over service composition, we introduce a configurable stateful-stateless weighting factor (*sfsl*). This parameter allows users to adjust the relative importance of stateful and stateless nodes, for example to prioritize stateful nodes in storage focused benchmarking scenarios. The *sfsl* value specifies the relative weight assigned to stateful services relative to stateless ones. During scoring, each node is assigned a weight of 1.0 (default) if it is stateless, and the user-defined *sfsl* value (default is set to 1.0) if it is stateful. This weighted score is then multiplied by the node's popularity to compute the final NIS.

The **Request Importance Score (RIS)** then aggregates node-level scores to quantify the importance of each request trace. Let R_j be the set of nodes in the j-th request call graph, and let $i \in R_j$ represent a node within that request. We define the Request Importance Score (RIS) as the normalized sum of NIS values over all nodes in the request: $\text{RIS}(T_j) = \frac{1}{|R_j|} \sum_{i \in R_j} \text{NIS}(i)$.

This normalized formulation ensures that request call graphs of varying depths can be fairly compared. Requests traversing more critical or frequently used services, as captured by higher NIS values, are assigned higher RIS scores. These importance scores guide the sampling process, ensuring that the downscaled trace retains the most topologically representative portions of the original trace.

3.3.2 Sampling Phase

The goal of the sampling phase is to extract a reduced subset of nodes and request call graphs that best represent the original trace (according to the previously defined metrics) while adhering to user-specified scale constraints, namely, the number of services (n) and request call graphs (t). This phase ensures that the downscaled trace remains representative in terms of both critical services and request path diversity, as determined by the scoring phase.

To retain the most important services, we begin by selecting nodes using priority sampling based on the Node Importance Score (NIS). In particular, service nodes are ranked in descending order of NIS, and the top n nodes are selected.

Next, we select t request call graphs using stratified sampling, guided by the Request Importance Score (RIS). In particular, we partition request traces into strata based on their call depth. From each stratum, a proportionate number of request call graphs are sampled, such that the overall call-depth distribution mirrors that of the original trace. Within each stratum, requests are prioritized by RIS, ensuring that the most representative traces are selected. This achieves two important goals: first, to preserve the distribution of request path lengths (i.e, call depths) and second, to favor request call graphs that traverse critical services.

At this stage, the combined set of unique nodes in the t sampled request call graphs may still exceed the node budget (n). This happens because request call graphs are selected based on RIS and call-depth, and are not constrained by the top n nodes ranked using NIS. To address this, we perform a final pruning step to ensure that all request call graphs contain only nodes from the chosen set of n services.

3.3.3 Pruning Phase

The final phase of the downscaling process refines the trace by enforcing the node constraint n on the sampled traces t. To this end, the nodes not included in top n nodes must be pruned from the selected request call graphs.

A challenge that arises is that naively removing unselected nodes along with their associated edges risks fragmenting the request call graphs into disconnected subgraphs or introducing orphaned service nodes. In these cases, the request execution flow would be interrupted. To prevent this, we implement a removal strategy that checks if the node to be removed is a bridge node, i.e., nodes whose removal would fragment the trace into disjoint components. If so, then we modify the graph by reconnecting their predecessors to

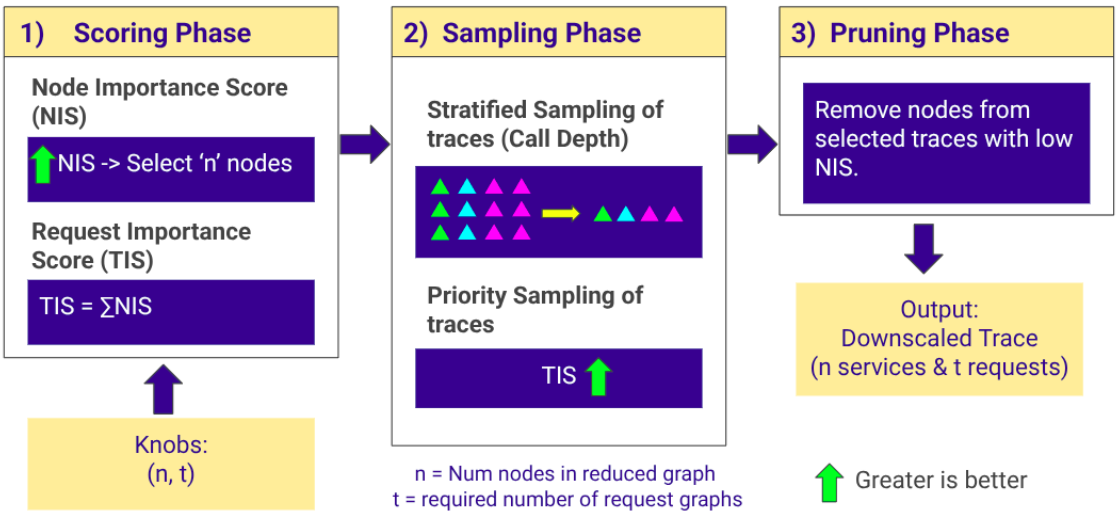

Figure 3.2: Downscaler execution flow.

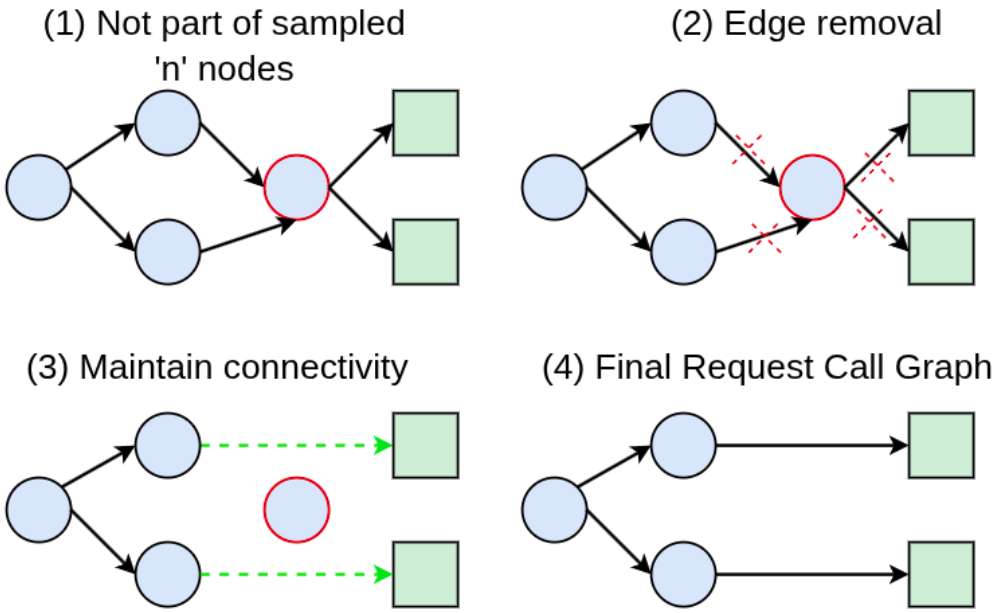

Figure 3.3: Bridge node pruning.

their successors, effectively bypassing the node while preserving the original call flow. This ensures that the trace graph remains connected and that the original sequence of service interactions is preserved after pruning (refer to fig. 3.3).

The final output is a downscaled trace that adheres to both the node budget n and the required request count t, while preserving the execution logic and topological properties of the original trace.

4

Mewbie Implementation: Benchmark Generation & Execution

The Mewbie implementation is organised into two distinct phases: benchmark generation (section 4.1) and benchmark execution (section 4.2). In the generation phase, Mewbie takes microservice traces (downscaled) and user defined inputs to construct a realistic, replayable and deployable benchmark. In the execution phase, the generated benchmark is automatically deployed in a controlled environment where request workloads are replayed and smetrics logged for further analysis of system behaviours under the user defined workloads. Together, these phases enable repeatable, customizable, and scalable benchmarking of microservice architectures. This addresses Research Question 2 (RQ2), which focuses on designing and implementing a benchmarking framework with support for scalability, customizability, and automated deployment.

4.1 Benchmark Generation

Generating a realistic benchmark from production traces presents two challenges. First, traces typically lack critical service-level information such as the type of service and inter-service call information. Second, replicating the logic and execution behavior of production services is infeasible without access to application code.

To address these challenges, Mewbie introduces two tightly connected components in this phase: the trace enrichment tool and the skeletal application. The trace enrichment tool (§4.1.1) augments raw traces with missing semantic information to produce enriched request call graphs that encode the semantics and execution flow of each request. Enriched

request call graphs form the benchmark workload. Based on the structure of the down-scaled trace and the semantics captured in the enriched request call graphs, Mewbie then constructs a skeletal application (§4.1.2): a lightweight, containerized version of the service logic. The skeletal application, when activated by the trace's external user requests, will then reproduce the trace's observed microservice calls.

4.1.1 Trace Enrichment

To convert a microservice trace into a runnable benchmark, Mewbie must first bridge the gap between what traces provide and what replaying them requires. Publicly available traces typically omit critical semantic details needed for realistic replay (4). These include service roles (e.g., SQL/NoSQL datastores, stateless logic), operation types (e.g., read, write), and parameters like the payload size.

Mewbie's *trace enrichment tool* fills in these semantic details by augmenting the trace at two levels:

1. **Service Level:** At the service level (i.e., at each of the call graph nodes), our tool defines the configuration of specific service types, to enable testing different settings, workloads, and deployment environments. In particular, users can configure parameters for these services, especially for datastores, including the database type (e.g., Redis, MongoDB), the number of records, replication factors, and consistency models.

2. **Call Level:** Mewbie generates a call object for each interaction between services. This object encodes execution-level details such as the operation type (read or write), object identifiers, and payload sizes. This detailed information enables trace-driven evaluations that reflect how requests would propagate and behave in target microservice environments.

Datastore Call Properties. Leaf nodes in request call graphs typically correspond to datastore accesses and are treated with additional semantics. For these calls, Mewbie specifies the operation types and the key-value pair being accessed. Keys are drawn from a user defined distribution, such as Zipfian or uniform, to model realistic access skew. Values are randomly generated based on a user-specified object size distribution.

Service Processing Time. To approximate the latency behavior of real systems, Mewbie assigns processing delays to stateless services based on observed execution latencies in

the original trace. Specifically, for each service type, we extract the distribution of processing times by measuring the duration between a service receiving a request and initiating the next outbound call. These delays are collected across all occurrences of the service in the trace.

Then, to construct the respective distribution, we compute latency percentiles at fixed 5% intervals from the 5th up to the 95th percentile and sample from this distribution during trace enrichment. This enables Mewbie to reproduce the timing variability observed in production systems instead of relying on fixed delays for all services.

Benchmark Knobs. This phase exposes knobs for defining service roles and tuning request behavior. Users assign roles such as stateless, cache, or datastore, which determine the available configuration options. Stateless services support control over payload sizes and call execution modes (synchronous or asynchronous). For caches and datastores, users can specify the engine (e.g., Redis, MongoDB), storage size, record count, object size distribution, and read/write ratios. Datastores can be tuned to adjust replication and consistency levels. These knobs allow users to tailor trace semantics to reflect target workloads and skeletal application behaviors.

4.1.2 Skeletal Application

The skeletal application is a lightweight, configurable replica of the processing logic of the different microservices, closely replaying the computations that generated trace. This enables realistic benchmarking without requiring access to the original applications. Generating the skeletal application begins with mapping each service node in the downscaled and enriched trace to a corresponding containerized component. These components are divided into stateless nodes, which simulate service processing latency based on the latency distribution extracted from the original trace (as described above), and stateful nodes such as databases, which are not simulated but instead implemented through accesses to real systems (e.g., Redis, PostgreSQL).

Modular Design. An important feature of our design is its modularity, which is essential for allowing different configurations or replacing service components, such as datastores, message queues, and load balancers. In particular, users can easily swap service components (e.g., replacing Redis with MongoDB) or adjust service configurations like replication factors, consistency settings, or cache size to evaluate different architectural choices. This modularity facilitates targeted benchmarking of individual or combinations of services, making Mewbie adaptable to diverse testing scenarios and evolving architectural patterns. The respective technical implementation details are further elaborated in section 5.1.

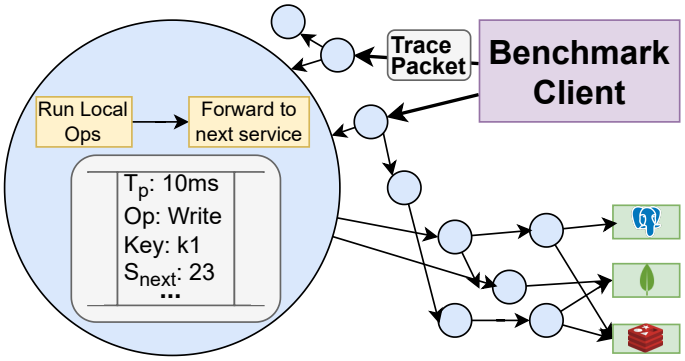

Figure 4.1: Stateless processing.

Request Flow. To understand how the sequence of actions triggered by a request is implemented, recall that each request in Mewbie is encapsulated in an enriched request call graph generated by the trace enrichment tool (section 4.1.1). This encodes all the information required to simulate the sequence of interactions across service nodes that are triggered by a given request, namely it consists of a sequence of call objects, each corresponding to a service invocation in a request call graph.

In this context, each call object in the enriched request call graphs can be of two types: stateless or stateful. Stateless call objects specify parameters such as the communication mode (sync/async), or the request processing time (T_p), indicating how long a stateless node should simulate processing delays before performing the call to the downstream node. In turn, stateful call objects specify datastore-specific information, namely the database operation type (Op) and the key (Key) to be accessed. These parameters are then passed to the shim layer associated with the respective datastore or message queue to execute the operation. Each call object also includes a next-call field (S_{next}), instructing the skeletal application on the subsequent service or set of services to invoke, thus defining the structure of the request flow.

The implementation of each simulated stateless service node is a lightweight HTTP server capable of receiving and processing requests. Upon receiving the request (enriched request call graph), the service performs a controlled delay using the sleep function to simulate processing time (refer to 4.1). It extracts the corresponding call objects and determines whether to make a synchronous or asynchronous call. If the downstream target is another stateless service, the enriched request call graph is forwarded as an HTTP request to that service. If the next call targets a stateful service (e.g., a database), the node executes the appropriate operation using shim logic and metadata provided in the call object. All

Table 4.1: Configuration knobs in Mewbie, organized by group.

Knob Group	Parameter
Trace Downscaling	Number of services (node budget) Number of request traces (call graphs)
Trace Enrichment	Service role (stateless/cache/datastore) Communication mode (sync/async) Payload size distribution Datastore engine & replicas Number records per datastore Object size distribution Operation type (read, write) Datastore consistency model
Benchmark Execution	Request issuing pattern Number of host machines Hot vs. non-hot node resource CPU & Memory per service

stateless services are pre-initialized with the necessary shims and client bindings to ensure seamless interactions with real datastores.

Logging. During execution, stateless service nodes collect and log detailed telemetry for each request, including timestamps, inter-service latencies, and throughput metrics. These measurements are used to compute performance indicators such as request completion time and system-wide throughput, as described in section 2.3.

4.2 Benchmark Execution

After downscaling the original trace, enriching the trace, and generating a skeletal application, Mewbie transitions to the execution phase. Deploying and orchestrating a large number of services environment is complex and leads to various different resource bottlenecks, port exhaustion, or performance variability, which can obscure the effects of architectural decisions or lead to failed executions.

To address these challenges, and enable reliable and reproducible execution at scale Mewbie automates the entire execution pipeline. It provides a deployment module that provisions and configures all service containers, a benchmark client that replays enriched request call graphs to drive the system, and a shim layer that facilitates communication with the datastore engine. Additionally, Mewbie incorporates fine-grained instrumentation within the skeletal application, allowing it to collect key performance metrics such as request completion time and system throughput.

Together, these components enable Mewbie to run realistic benchmarks at scale, while enabling a detailed analysis of system behavior under controlled conditions.

4.2.1 Deployment Infrastructure

Mewbie provides an automated deployment infrastructure that provisions and configures all service containers based on the enriched trace and skeletal application. The deployment module identifies unique service nodes and initializes stateful components using user-provided container images. This allows the benchmark to run seamlessly across single or multi-host environments, regardless of scale.

Even though Mewbie uses lightweight containerized components to simulate services, executing benchmarks at scale introduces practical challenges – especially in resource constrained settings. In particular, the high request volume, shared compute resources, and uneven service load can quickly lead to various different resource bottlenecks.

One particularly pressing challenge involves *hot nodes*: services that handle a disproportionately large number of requests and can become bottlenecks if not adequately handled. During its trace enrichment phase, Mewbie automatically identifies these heavily utilized nodes and allows users to assign additional CPU and memory via separate resource knobs for hot and non-hot services. With this separation, Mewbie enables users to allocate additional CPU and memory where needed, preventing artificial resource starvation and ensuring that high-traffic services can sustain large volumes of concurrent connections. This targeted approach, combined with optimized networking configurations, helps maintain consistent performance across the entire microservice application.

Managing High-Throughput Service Calls. In microservice benchmarks, each service node can be involved in thousands of calls per second, especially for stateless nodes that act as intermediaries in many request call graphs. To support such high throughput communication without overwhelming the network stack, Mewbie configures each container with elevated network limits. These include file descriptor counts (`ulimit`), expanding TCP port range (`ip_local_ port_range`), and enabling TCP reuse (`tcp_tw_reuse`). These tweaks help avoid port exhaustion and reduce socket wait times, making the system capable of handling a high volume of concurrent connections.

Resource Aware Scheduling. In microservice systems, some services are involved in a much higher number of requests than others, placing them under significantly higher load. We refer to these frequently invoked services as hot-nodes, as they tend to become performance bottlenecks. During the enrichment phase (refer to section 4.1.1), Mewbie analyses the trace to identify such hot-nodes.

To account for this during deployment, Mewbie exposes two separate resource configuration knobs: one for hot nodes and another for non-hot nodes. This allows users to allocate additional CPU and memory to heavily utilized services, reducing the risk of performance bottlenecks caused by resource starvation. As a result, the system's performance reflects architectural design choices rather than infrastructure misconfiguration.

Benchmark Client. To execute the benchmark, Mewbie includes a client that replays the workload described in the generated enriched request call graphs. Each request is injected into its designated entrypoint service, initiating its execution in the skeletal application. The client supports a range of request arrival patterns, but users can also define custom inter-arrival distributions to model specific traffic behaviors. This flexibility allows Mewbie to simulate diverse load conditions, enabling evaluation of system responsiveness and performance under target scenarios.

Benchmark Knobs. Mewbie exposes a set of knobs to control runtime behavior and resource allocation during benchmark execution. Users can assign services to specific host machines and configure CPU and memory limits for each container applying different settings for hot and non-hot nodes as needed. On the workload side, the benchmark client supports configurable request arrival patterns, including uniform, ramp-up, and custom inter-arrival distributions, allowing users to model a uniform or a gradual increase or decrease in request rate over time. These knobs collectively enable realistic deployment scenarios and help isolate performance effects tied to architectural and resource configurations.

4. MEWBIE IMPLEMENTATION: BENCHMARK GENERATION & EXECUTION

5

Evaluation

We evaluate Mewbie in terms of the core requirements outlined in section 2.1, examining whether it meets these criteria and aligns with the broader goal of providing a flexible microservice benchmarking solution. Our analysis is structured around three evaluation questions:

1. **Can Mewbie downscale large-scale microservice traces while preserving essential topological properties?**
2. **Can Mewbie evaluate the performance of microservice deployments under different architectural decisions?**
3. **Can Mewbie benchmarks be easily customized in terms of adjusting each service's internal configurations?**

This addresses Research Question 3 (RQ3), which investigates the effectiveness of the Mewbie benchmark in revealing architectural tradeoffs and quantifying the impact of service-level configuration changes in microservice systems.

5.1 Experimental Setup

Trace Downsampling. Our experiments utilize production microservice traces from Alibaba's clusters (4). We randomly sampled 650k request graphs from the dataset to obtain a manageable working set for analysis and benchmarking. To address structural inconsistencies, we applied the Casper tool (51), followed by additional cleanup to handle remaining issues such as missing node labels and absent entrypoint information. We patched these cases using randomized labelling and by assigning entrypoints from the top ten most frequent entrypoint services.

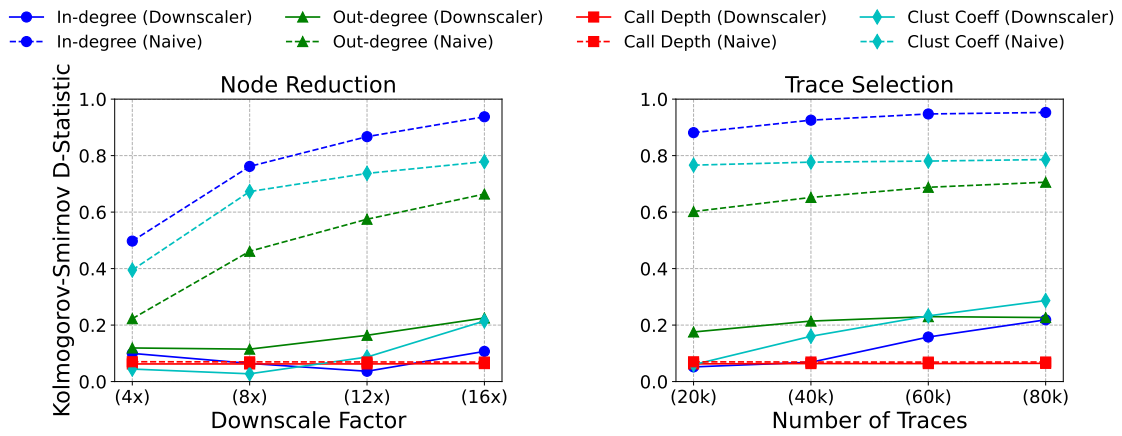

Figure 5.1: Mewbie downscaling process evaluation: KSDS evaluation (lower is better).

The resulting cleaned trace contained 650k request call graphs with 8000 unique microservices. Increasing the sample size is straightforward and impacts both the pre-processing and downscaling time. Downscaling was executed on a single machine with 20 CPU cores and 251 GB RAM.

Mewbie Execution. Experiments ran on a four-node cluster, each with 20 cores and 251 GB RAM. Unless stated otherwise, our evaluations used a downscaled trace with a node budget of 500 microservices (n) and 250k request call graphs (t). This corresponds to a 16x reduction from the sampled trace. We select the top 10 stateless and top 10 stateful nodes (based on request frequency) as hot nodes, as described in section 4.2. Hot stateful nodes are allocated 1 CPU core and 4 GB of memory, hot stateless nodes receive 6 cores and 6 GB, while non-hot stateful and stateless nodes are assigned 0.5 cores / 4 GB and 1 core / 4 GB, respectively. We configured the trace enrichment knobs to use a fixed payload size of 15 KB and a uniform access pattern across 10,000 objects per datastore.

5.2 Can Mewbie downscale large-scale microservice traces while preserving topological properties?

Our first experiment evaluates whether Mewbie's downscaling process can significantly reduce trace size while preserving key topological properties, outlined in table 3.1.

Baseline. To assess the effectiveness of our trace downscaling technique, we compare it against both the original trace and a simple sampling baseline. This baseline method selects a subset of t request call graphs at random from the original trace dataset. In the interest of adhering to the user defined node budget, each node within these selected request call

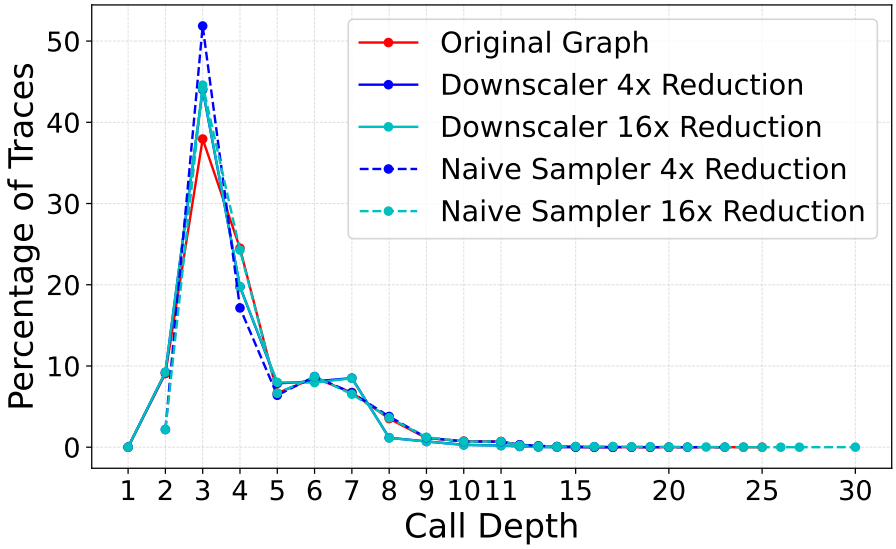

Figure 5.2: Mewbie downscaling process evaluation: Call depth distribution comparison.

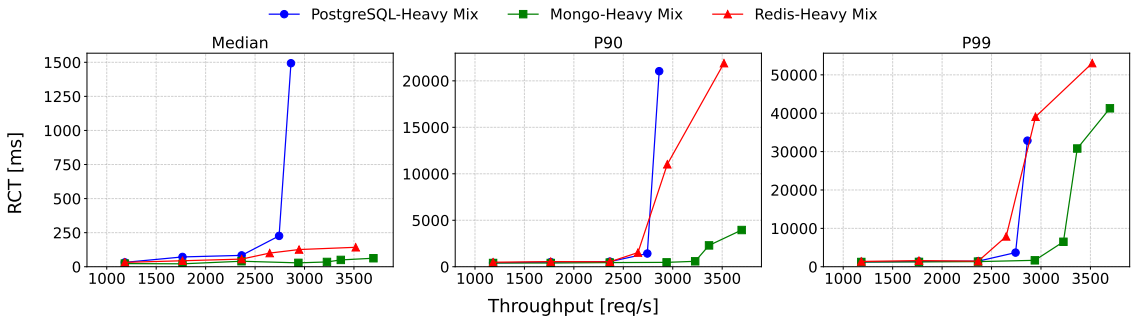

Figure 5.3: Mewbie output for application throughput vs. request completion time under different datastore mixtures.

graphs is then reassigned to new node identifiers. This straightforward approach provides a basic form of trace reduction for comparative evaluation.

Downscaling Factor. A Mewbie user starts by simply specifying the *desired number of unique nodes* (n), typically based on the available infrastructure, allowing the user to tailor the downscaling to its resource availability. We evaluate the impact of the downscaling process by analyzing how different reduction factors affect key topological properties. Specifically, we consider downscaled traces at 4x, 8x, 12x, and 16x reductions, which reduce the original set of 8000 services into, 2000, 1000, 665, 500 nodes, respectively.

To assess how well the distribution of several topological properties are preserved during downscaling, we use the Kolmogorov–Smirnov D-statistic (*KSDS*) (52), which measures the maximum difference between two distributions, making it well-suited for comparing

structural similarity between the original and the downscaled graphs (50). fig. 5.1 presents KSDS results across four important graph topological metrics: call-depth distribution, in-degree & out-degree distribution, and clustering coefficient, for both Mewbie's downscaler and the sampling baseline, evaluated across downscaling factors from 4x to 16x.

Across all metrics except call-depth, Mewbie consistently achieves substantially lower KSDS values compared to the baseline, thus indicating a closer alignment to the topological structure of the original trace. This trend is particularly evident in the in-degree and out-degree distributions, where the downscaler achieves up to 3-4x lower divergence across all factors. Mewbie also maintains strong fidelity for the clustering coefficient, with KSDS values remaining as low as 0.21, even at 16x reduction. In contrast, the sampling baseline rapidly deteriorates, reaching KSDS values as high as 0.79.

The sampling baseline performs comparably in call-depth, which it tends to preserve by virtue of randomly sampling full request call graphs with no pruning of service nodes. Still, as shown in fig. 5.2, simple sampling introduces sharp deviations, particularly at low downscaling factors. For example, the naive sampling technique exhibits a skew of $\sim$14% at depth 3 at 4x reduction, whereas the downscaler maintains a closer approximation, with a maximum skew of $\sim$6%. To revisit Research Question 1 (RQ1); *How to generate realistic and usable benchmarks and workloads from production-scale microservice traces through systematic enrichment and transformation?*, our results show that our downscaler approach stands out by simultaneously optimizing multiple topological metrics, unlike simpler strategies that neglect the effect of sampling on other key metrics. As a result, we yield significantly higher overall fidelity to the original production-grade traces.

5.3 Can Mewbie evaluate the performance of microservice deployments?

We evaluate Mewbie's capability to benchmark the performance impact of different architectural decisions on microservice deployments, particularly regarding various *datastore mixtures*. The term "datastore mixture" refers to the percentage of different datastore systems within a microservice-based application.

Modern microservice applications frequently use diverse datastores: caches for low-latency access, relational databases for structured queries, and NoSQL stores for scalability and flexibility (11, 12). Recent analysis shows that 42.35% of cloud architectures used two or more storage services (10), highlighting the prevalence of datastore mixtures in practice. These combinations significantly impact latency, throughput, and resource

contention. Mewbie enables precise control to generate and evaluate such architectural choices (10).

To demonstrate this capability, we generate and evaluate three datastore mixtures that reflect common deployment patterns: one predominantly utilizing PostgreSQL, another favoring MongoDB, and a third favoring Redis. In all cases, the predominant datastore represents 80% of the storage systems, and the other represent 10%. The experimental results for these mixtures are depicted in fig. 5.3, using the two essential metrics introduced in section 4.2: request completion time (RCT), the end-to-end time to execute a complete request chain, and application throughput, measured from the client's perspective. To precisely characterize these tradeoffs, we plot throughput-latency curves, where we increase the load that clients submit in different runs, and each point plots the measured latency and throughput for that load. For this experiment, all service calls are synchronous, meaning the client actively waits for the entire downstream chain of requests to finish before proceeding.

PostgreSQL-Heavy Mix. This mixture represents applications that require structured data storage and ACID compliance. Such a mixture is ideal for applications like online banking and e-commerce platforms, where transaction integrity and structured data storage is crucial (53). As shown in fig. 5.3, this mixture yields the highest median and p90 request completion times (RCT) and lowest overall throughput among the three mixtures. This is expected, given the additional overhead associated with transaction coordination and complex relational operations in PostgreSQL, which increases latency under load. Interestingly, the p99 latency for this mix remains comparatively stable, indicating lower variability. This suggests that while PostgreSQL-heavy deployments may trade off average-case responsiveness, they offer greater predictability at the tail.

Mongo-Heavy Mix. This mixture represents workloads that prioritize flexible schema design and handling unstructured data, and is ideal for applications that involve real-time analytics, and media streaming features (53). As shown in fig. 5.3, this mixture yields consistently better throughput across all percentiles, outperforming the other two mixtures. It shows a max throughput of 3400 requests per second (rps). These results indicate that Mongo-heavy deployments are well-suited for our evaluation's workload characteristics, particularly where flexible data handling and high throughput are prioritized over strict consistency or minimal tail latency.

Redis-Heavy Mix. This mixture is representative of workloads emphasizing fast data retrieval and minimal latency, like social media platforms (54). As shown in fig. 5.3, the Redis-heavy mixture delivers the lowest median request completion time (RCT) among

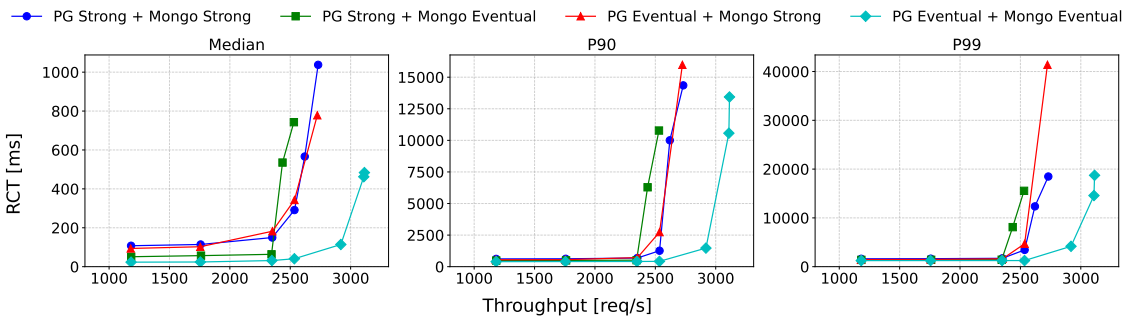

Figure 5.4: Consistency Models: Application throughput vs. Request Completion time.

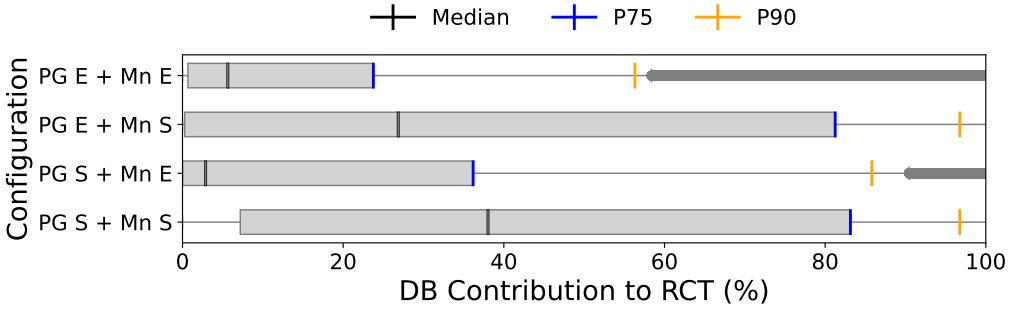

Figure 5.5: Backend pressure at ∼2500 rps (PG=PostgreSQL, Mn=MongoDB, S=Strong consistency, E=Eventual consistency).

all three configurations, confirming Redis's strength in quickly serving latency-sensitive requests. However, this mixture performance comparably to the PostgreSQL-mix at the tail, suggesting increased latency variability under load. For use cases that prioritize low-latency responsiveness and can tolerate occasional tail latency spikes, Redis-heavy deployments offer clear performance advantages.

5.4 Can Mewbie benchmarks be customized, in terms of adjusting service internal configurations?

Microservice systems are highly configurable and services are frequently tuned to balance trade-offs between responsiveness, consistency, and scalability (2). A key configuration point is the datastore consistency model, which defines how data is maintained across nodes and replicas in a distributed architecture. The consistency model determines the trade-off between data staleness and performance (55, 56). In this section, we evaluate Mewbie's ability to benchmark the impact of consistency tuning at the storage layer. We aim to

demonstrate that Mewbie enables users to isolate, explore, and quantify the performance effects of tuning such parameters without changing the overall architecture.

Additionally, by extracting backend pressure, i.e., how much the storage layer contributes to the end-to-end request latency, Mewbie helps provide deeper visibility into how consistency configurations impact performance. This helps uncover non-obvious behaviors, for example, when relaxing consistency in one service increases load or delays in others, depending on how services are ordered or interact in the request call graph.

Datastore Consistency Model. We build on the previous datastore mixture example by further adjusting configuration options in both PostgreSQL and MongoDB, this time focusing on variations in *consistency* models. Achieving an optimal balance between performance and consistency is notoriously challenging in distributed systems (57), and can significantly affect overall application behavior (58, 59).

For example, banking services require strong consistency to ensure correct transactions (57), whereas social media platforms typically prioritize low latency to improve responsiveness (60). In a microservice environment, these trade-offs become even more complex due to the variety and interdependence of specialized datastores. In modern microservice environments, these trade-offs often co-exist within the same application. Different services may have varying consistency requirements based on their role, correctness guarantees, performance constraints.

To model this diversity, we evaluate four *consistency mixtures*, analogous to our previous datastore mixtures, but incorporating consistency parameters instead. We maintain a 50%-50% distribution of services between PostgreSQL (PG) and MongoDB, altering only the datastore consistency levels within each:

- **PG & MongoDB Strong:** PostgreSQL is configured for synchronous replication, ensuring that writes are acknowledged only after being flushed to a remote standby. MongoDB employs linearizable consistency (`readConcern: majority`), ensuring that all reads reflect the most recent writes.

- **PG Strong & MongoDB Eventual:** PostgreSQL uses the same configuration as above, while Mongo is set to eventual consistency using (`readPreference: secondary`), reducing coordination overheads for reads.

- **PG Eventual & MongoDB Strong:** PostgreSQL uses a relaxed isolation level (async replication) paired with MongoDB using strong consistency.

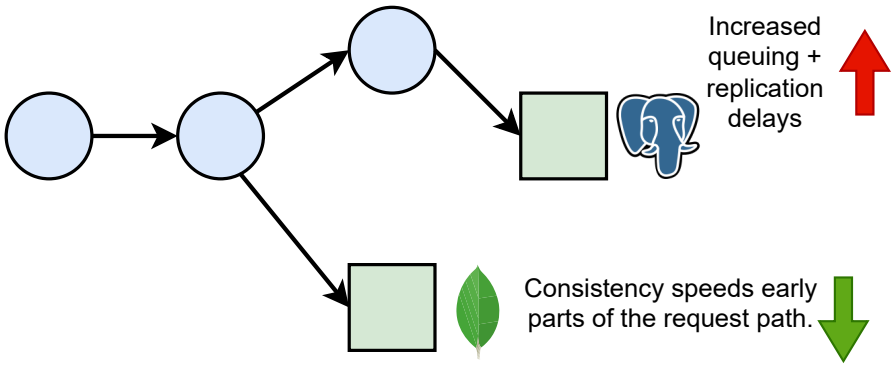

Figure 5.6: Increased queuing at Postgres in deeper parts of the request call graph.

- **PG & MongoDB Eventual:** Both PostgreSQL and MongoDB use eventual consistency to maximize performance and availability, at the cost of potential data staleness.

As shown in fig. 5.4, beyond approximately 2400 requests per second (rps), distinct trends begin to emerge across configurations. The PG Eventual + Mongo Strong configuration performs similarly, with only marginal improvements in median and p90, suggesting that MongoDB's strong consistency shapes the tail latency significantly. In contrast, the fully eventual setup (PG Eventual + Mongo Eventual) achieves the highest throughput and the lowest median and p90 latency. To better understand these effects, we examine backend pressure. As shown in fig. 5.5, the fully strong setup shows consistently high backend pressure, while the fully eventual configuration, despite its lower median, exhibits a significantly longer tail. This suggests that some requests still experience prolonged delays in the storage layer under relaxed consistency.

Interestingly, the PG Strong + Mongo Eventual configuration performs worse than both Mongo-strong setups, despite relaxing consistency on MongoDB. While it shows lower backend latency contribution (median to p90) than the fully strong configuration, it hits a throughput wall earlier. We conjecture that this is maybe due to MongoDB appearing earlier in the request path, its faster completion under relaxed consistency leads to higher request rate at Postgres, which remains strongly consistent. This increased downstream pressure on Postgres, combined with its synchronous replication overhead, likely leads to increased queueing, contention, and thus leading to earlier saturation. This showcases that performance degradation can emerge not just from individual service behavior, but from

how those behaviors interact within the structure of the request paths – a cascading effect that is captured by Mewbie.

These observations highlight a key strength of Mewbie: it enables users to tune and evaluate service-level configurations considering the causal structure of real-world traces.

6

Related Work & Conclusion

6.1 Related Work

Existing solutions for exploring and evaluating microservice architectural decisions fall into two categories: simplistic demo applications, or fixed-scale benchmarks. Neither approach successfully mirrors the architectural complexity and scale found in production environments, where companies like Alibaba (4), Uber (2), and others (3, 19, 61) operate in the range of thousands of microservices. table 1.1 summarizes existing systems across four critical dimensions, each aligned with Mewbie's requirements detailed in section 2.1.

Demo Applications. Perhaps the most common approach for experimenting with microservices tools and architectural decisions is through demo applications. Major industry players with microservice-related offerings frequently provide their own reference implementations, with Google's Online Boutique (21) (often used to showcase gRPC or Kubernetes) standing as the most prominent example, alongside similar applications from Microsoft (22), Jaeger (24), and Istio (23).

Although these demo applications are widely used to illustrate microservice concepts because of their straightforward setup and deployment, they do not capture the full complexity or performance demands of real-world systems. Typically, they include only a few dozen services, lack realistic workload patterns, and require extensive customization to suit specific research or testing needs.

Benchmarks. In response to the need for larger microservice testbeds, several benchmarks have been introduced. Most of these benchmarks bundle together a set of applications, each geared toward a particular domain or workload. For example, TrainTicket (26) focuses on tracing and debugging, μSuite (15) focuses on tail latency evaluation, and DeathStarBench (14) provides a suite of cloud-native workloads to study the performance

and architectural implications of microservices across different layers of the cloud stack. However, all of these are fundamentally tied to predefined applications, resulting in a small and fixed scale deployment, and with limited customizability. This lack of flexibility prevents users from tailoring scale, workload mixes, and service configurations to their specific needs. Furthermore, as the microservice landscape rapidly evolves with new technologies and frameworks, these benchmarks may struggle to keep pace.

In short, existing benchmarks and demo applications fail to capture the complexity and demands of real-world microservice environments. They cannot be scaled to realistic sizes, lack the versatility to adapt to different domains, and offer little flexibility for fine-tuning parameters. As a result, researchers and practitioners struggle to evaluate whether a given architecture, datastore, or design and configuration choice can handle production-scale workloads.

In response, Mewbie tackles these limitations through a domain-agnostic skeleton application that users can easily refine with various knobs, enabling both rapid prototyping and more detailed customization. It is designed to scale dynamically – up to near-production-level sizes – while providing architectural patterns that mirror those of real-world applications. For these reasons, Mewbie is an orthogonal contribution that could prove useful to a vast body of work on techniques for improving the design, implementation, and deployment of microservice systems, such as Antipode (62), Blueprint (63) and others (37, 38, 49, 64, 65, 66, 67, 68).

6.2 Conclusion

Microservice architectures offer scalability and agility, but their complexity and scale make it difficult to assess how specific design and architectural choices impact performance. To evaluate and test such systems, developers today are limited to small-scale applications that fall short of representing real-world deployment challenges.

We propose Mewbie an open-source benchmark that (1) generates an application architecture based on a real-world traces – thereby capturing realistic service-to-service communication patterns; (2) can scale to the level of resources available, from a handful to thousands of machines; and (3) requires minimal effort to adapt, making it easy to experiment with various architectural and design options, whether evaluating datastores, or other service-level parameters.

We believe Mewbie will play a key role in advancing the design, evaluation, and transparency of microservice-based systems, providing the community a practical and powerful tool for both research and real-world deployment testing.

References

[1] 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. 1

[2] 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. 1, 2, 3, 12, 14, 15, 34, 39

[3] f5.com. **Adopting Microservices at Netflix: Lessons for Architectural Design**, 2015. [Accessed 26-08-2024]. 1, 3, 39

[4] 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. 1, 2, 3, 6, 9, 13, 15, 22, 29, 39

[5] Cesare Pautasso, Olaf Zimmermann, Mike Amundsen, James Lewis, and Nicolai Josuttis. **Microservices in Practice, Part 1: Reality Check and Service Design**. *IEEE Software*, **34**(1):91–98, 2017. 1

[6] Grzegorz Blinowski, Anna Ojdowska, and Adam Przybyłek. **Monolithic vs. microservice architecture: A performance and scalability evaluation**. *IEEE access*, **10**:20357–20374, 2022. 1

[7] Netflix Technology Blog. **Netflix video quality at scale with cosmos microservices**, Nov 2021. 1

REFERENCES

[8] Shutian Luo, Huanle Xu, Kejiang Ye, Guoyao Xu, Liping Zhang, Guodong Yang, and Chengzhong Xu. **The power of prediction: microservice auto scaling via workload learning**. In *Proceedings of the 13th Symposium on Cloud Computing*, pages 355–369, 2022. 1

[9] Ajit Puthiyavettle Senthil Kumar. **Architecting a highly available serverless, microservices-based Ecommerce site**. 1

[10] Sambhav Satija, Chenhao Ye, Ranjitha Kosgi, Aditya Jain, Romit Kankaria, Yiwei Chen, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Kiran Srinivasan. **Cloudscape: A Study of Storage Services in Modern Cloud Architectures**. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*, pages 103–121, Santa Clara, CA, February 2025. USENIX Association. 2, 32, 33

[11] Rohan Basu Roy and Devesh Tiwari. **StarShip: Mitigating I/O bottlenecks in serverless computing for scientific Workflows**. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, **8**, 2 2024. 2, 32

[12] Tao Li, Yongkun Li, Wenzhe Zhu, Yinlong Xu, and John C. S. Lui. **MinFlow: High-performance and Cost-efficient Data Passing for I/O-intensive Stateful Serverless Analytics**. In *22nd USENIX Conference on File and Storage Technologies (FAST 24)*, pages 311–327, Santa Clara, CA, February 2024. USENIX Association. 2, 32

[13] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. **Benchmarking cloud serving systems with YCSB**. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010. 2, 7

[14] 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. 2, 3, 10, 39

[15] Akshitha Sriraman and Thomas F Wenisch. **μ suite: a benchmark suite for microservices**. In *2018 ieee international symposium on workload characterization (iiswc)*, pages 1–12. IEEE, 2018. 2, 3, 10, 39

[16] Joakim Von Kistowski, Simon Eismann, Norbert Schmitt, André Bauer, Johannes Grohmann, and Samuel Kounev. **Teastore: A micro-service reference application for benchmarking, modeling and resource management research**. In *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 223–236. IEEE, 2018. 2, 3, 10

[17] Takanori Ueda, Takuya Nakaike, and Moriyoshi Ohara. **Workload characterization for microservices**. In *2016 IEEE international symposium on workload characterization (IISWC)*, pages 1–10. IEEE, 2016. 2, 10

[18] Mathias Schwarz. **Up: Portable Microservices Ready for the Cloud**, 2023. [Accessed 18 Apr 2025]. 3

[19] Ranju R. **Microservices at eBay-What it looks like today**, 2021. [Accessed 18 Apr 2025]. 3, 39

[20] Hrishikesh Barua. **How Monzo Isolated Their Microservices Using Kubernetes Network Policies**, 2019. [Accessed 18 Apr 2025]. 3

[21] Google. **Google Microservice Demo: Online Boutique**, 2025. [Accessed 18 Apr 2025]. 3, 39

[22] Microsoft. **DotNet eShop**, 2025. [Accessed 18 Apr 2025]. 3, 39

[23] Istio. **Istio BookInfo**, 2025. [Accessed 18 Apr 2025]. 3, 39

[24] Jaeger. **Jaeger HotRod**, 2025. [Accessed 18 Apr 2025]. 3, 39

[25] AcmeAir. **Acmeair**, 2015. [Accessed 18 Apr 2025]. 3

[26] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Wenhai Li, and Dan Ding. **Fault Analysis and Debugging of Microservice Systems: Industrial Survey, Benchmark System, and Empirical Study**. *IEEE Transactions on Software Engineering*, **22**(4):243–260, 2021. 3, 39

[27] Alexandru Iosup, Alexandru Uta, Laurens Versluis, Georgios Andreadis, Erwin Van Eyk, Tim Hegeman, Sacheendra Talluri, Vincent Van Beek, and Lucian Toader. **Massivizing computer systems: a vision to understand, design, and engineer computer ecosystems through and beyond modern distributed systems**. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 1224–1237. IEEE, 2018. 5

[28] Ian Sommerville. **Software engineering (ed.)**. *America: Pearson Education Inc*, 2011. 5

[29] Len Bass, Paul Clements, and Rick Kazman. *Software architecture in practice*. Addison-Wesley Professional, 2021. 5

[30] Andreas Loukas. **Graph reduction with spectral and cut guarantees**. *Journal of Machine Learning Research*, **20**(116):1–42, 2019. 6, 13

[31] Yuanyuan Tian, Richard A Hankins, and Jignesh M Patel. **Efficient aggregation for graph summarization**. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 567–580, 2008. 6, 13

[32] Andreas Loukas and Pierre Vandergheynst. **Spectrally Approximating Large Graphs with Smaller Graphs**. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, **80** of *Proceedings of Machine Learning Research*, pages 3237–3246. PMLR, 10–15 Jul 2018. 6, 13

[33] Alexandru Iosup, Fernando Kuipers, Ana Lucia Varbanescu, Paola Grosso, Animesh Trivedi, Jan Rellermeyer, Lin Wang, Alexandru Uta, and Francesco Regazzoni. **Future Computer Systems and Networking Research in the Netherlands: A Manifesto**. Workingpaper, May 2022. Position paper: 7 foundational research themes in computer science and networking research, 4 advances with outstanding impact on society, 10 recommendations, 50 pages. Co-signatories from (alphabetical order): ASTRON, CWI, Gaia-X NL, NIKHEF, RU Groningen, SIDN Labs, Solvinity, SURF, TNO, TU/e, TU Delft, UvA, U. Leiden, U. Twente, VU Amsterdam. 7

[34] Yu Gan, Mingyu Liang, Sundar Dev, David Lo, and Christina Delimitrou. **Sage: practical and scalable ML-driven performance debugging in**

microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 135–151, 2021. 9

[35] 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. 9

[36] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. **Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices**. In *Proceedings of the twenty-fourth international conference on architectural support for programming languages and operating systems*, pages 19–33, 2019. 9

[37] Jianshu Liu, Qingyang Wang, Shungeng Zhang, Liting Hu, and Dilma Da Silva. **Sora: A latency sensitive approach for microservice soft resource adaptation**. In *Proceedings of the 24th International Middleware Conference*, pages 43–56, 2023. 9, 40

[38] Kapil Agrawal and Sangeetha Abdu Jyothi. **Cooperative Graceful Degradation in Containerized Clouds**. In Lieven Eeckhout, Georgios Smaragdakis, Kaitai Liang, Adrian Sampson, Martha A. Kim, and Christopher J. Rossbach, editors, *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1, ASPLOS 2025, Rotterdam, The Netherlands, 30 March 2025 - 3 April 2025*, pages 214–232. ACM, 2025. 9, 40

[39] Donghyun Kim, Sriram Ravula, Taemin Ha, Alexandros G. Dimakis, Daehyeok Kim, and Aditya Akella. **Large Language Models as Realistic Microservice Trace Generators**. 12 2024. 10

[40] Polona Caserman, Michelle Martinussen, and Stefan Göbel. **Effects of end-to-end latency on user experience and performance in immersive virtual reality applications**. In *Entertainment Computing and Serious Games: First IFIP TC 14 Joint International Conference, ICEC-JCSG 2019, Arequipa, Peru, November 11–15, 2019, Proceedings 1*, pages 57–69. Springer, 2019. 12

REFERENCES

[41] Ioannis Arapakis, Xiao Bai, and B Barla Cambazoglu. **Impact of response latency on user behavior in web search**. In *Proceedings of the 37th international ACM SIGIR conference on Research & development in information retrieval*, pages 103–112, 2014. 12

[42] Ioannis Arapakis, Souneil Park, and Martin Pielot. **Impact of response latency on user behaviour in mobile web search**. In *Proceedings of the 2021 Conference on Human Information Interaction and Retrieval*, pages 279–283, 2021. 12

[43] Tatsushi Inagaki, Yohei Ueda, Moriyoshi Ohara, Sunyanan Choochotkaew, Marcelo Amaral, Scott Trent, Tatsuhiro Chiba, and Qi Zhang. **Detecting layered bottlenecks in microservices**. In *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*, pages 385–396. IEEE, 2022. 12

[44] Haryadi S Gunawi, Riza O Suminto, Russell Sears, Casey Golliher, Swaminathan Sundararaman, Xing Lin, Tim Emami, Weiguang Sheng, Nematollah Bidokhti, Caitie McCaffrey, et al. **Fail-slow at scale: Evidence of hardware performance faults in large production systems**. *ACM Transactions on Storage (TOS)*, **14**(3):1–26, 2018. 12

[45] Syed Akbar Mehdi, Cody Littley, Natacha Crooks, Lorenzo Alvisi, Nathan Bronson, and Wyatt Lloyd. **I Can't Believe It's Not Causal! Scalable Causal Consistency with No Slowdown Cascades**. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 453–468, 2017. 12

[46] Doug Beaver, Sanjeev Kumar, Harry C Li, Jason Sobel, and Peter Vajgel. **Finding a needle in haystack: Facebook's photo storage**. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, 2010. 12

[47] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. **Pivot tracing: Dynamic causal monitoring for distributed systems**. *ACM Transactions on Computer Systems (TOCS)*, **35**(4):1–28, 2018. 12

[48] OpenTelemetry. **OpenTelemetry: High-quality, ubiquitous, and portable telemetry to enable effective observability**, 2025. [Accessed 18 Apr 2025]. 13

[49] Shutian Luo, Jianxiong Liao, Chenyu Lin, Huanle Xu, Zhi Zhou, and Chengzhong Xu. **Embracing Imbalance: Dynamic Load Shifting among Microservice Containers in Shared Clusters**. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '25, page 309–324, New York, NY, USA, 2025. Association for Computing Machinery. 15, 40

[50] Jure Leskovec and Christos Faloutsos. **Sampling from large graphs**. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 631–636, 2006. 15, 16, 32

[51] Darby Huye, Lan Liu, and Raja R Sambasivan. **Systemizing and Mitigating Topological Inconsistencies in Alibaba's Microservice Call-graph Datasets**. In *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering*, pages 276–285, 2024. 29

[52] Frank J Massey Jr. **The Kolmogorov-Smirnov test for goodness of fit**. *Journal of the American statistical Association*, **46**(253):68–78, 1951. 31

[53] Avrilia Floratou, Jignesh M Patel, Willis Lang, and Alan Halverson. **When free is not really free: What does it cost to run a database workload in the cloud?** In *Topics in Performance Evaluation, Measurement and Characterization: Third TPC Technology Conference, TPCTC 2011, Seattle, WA, USA, August 29-September 3, 2011, Revised Selected Papers 3*, pages 163–179. Springer, 2012. 33

[54] Zeying Zhu, Yibo Zhao, and Zaoxing Liu. **In-Memory Key-Value Store Live Migration with NetMigrate**. In *22nd USENIX Conference on File and Storage Technologies (FAST 24)*, pages 209–224, 2024. 33

[55] Seth Gilbert and Nancy Lynch. **Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services**. *ACM SIGACT News*, **33**(2):51–59, jun 2002. 34

[56] Daniel Abadi. **Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story**. *Computer*, **45**:37–42, 1 2012. 34

[57] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. **Making Geo-Replicated Systems Fast**

as Possible, Consistent when Necessary. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI '12)*, 2012. 35

[58] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. **Don't settle for eventual: scalable causal consistency for wide-area storage with COPS**. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 401–416, 2011. 35

[59] Wojciech Golab, Muntasir R Rahman, Alvin AuYoung, Kimberly Keeton, and Xiaozhou Li. **Eventually consistent: Not what you were expecting?** *Communications of the ACM*, **57**(3):38–44, 2014. 35

[60] Haonan Lu, Kaushik Veeraraghavan, Philippe Ajoux, Jim Hunt, Yee Jiun Song, Wendy Tobagus, Sanjeev Kumar, and Wyatt Lloyd. **Existential consistency**. In *25th ACM Symposium on Operating Systems Principles (SOSP '15)*, 2015. 35

[61] Carlo Gutierrez. **Spotify Runs 1,600+ Production Services on Kubernetes**, September 2021. [Accessed 18 Apr 2025]. 39

[62] João Ferreira Loff, Daniel Porto, João Garcia, Jonathan Mace, and Rodrigo Rodrigues. **Antipode: Enforcing cross-service causal consistency in distributed applications**. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 298–313, 2023. 40

[63] 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. 40

[64] Mafalda Sofia Ferreira, João Ferreira Loff, and João Garcia. **Rendezvous: Where Serverless Functions Find Consistency**. In *Proceedings of the 4th Workshop on Resource Disaggregation and Serverless*, WORDS '23, page 51–57, New York, NY, USA, 2023. Association for Computing Machinery. 40

[65] Jacoby Johnson, Subash Kharel, Alan Mannamplackal, Amr S Abdelfattah, and Tomas Cerny. **Service Weaver: A Promising Direction for Cloud-native Systems?** *arXiv preprint arXiv:2404.09357*, 2024. 40

[66] Rajiv Nishtala, Vinicius Petrucci, Paul Carpenter, and Magnus Sjalander. **Twig: Multi-agent task management for colocated latency-critical cloud services**. In *Proceedings - 2020 IEEE International Symposium on High Performance Computer Architecture, HPCA 2020*, pages 167–179. Institute of Electrical and Electronics Engineers Inc., 2 2020. 40

[67] Rohan Basu Roy, Vijay Gadepally, and Devesh Tiwari. **DarwinGame: Playing Tournaments for Tuning Applications in Noisy Cloud Environments**. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pages 264–279. ACM, 3 2025. 40

[68] Divyanshu Saxena, William Zhang, Shankara Pailoor, Isil Dillig, and Aditya Akella. **Copper and Wire: Bridging Expressiveness and Performance for Service Mesh Policies**. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pages 233–248. ACM, 3 2025. 40