Characterizing Microservices and Accommodating Tracing Observability Loss

A dissertation

submitted by

Darby Huye

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

in

Computer Science

TUFTS UNIVERSITY

February 2026

Adviser: Raja R. Sambasivan

Abstract

Microservices have become the dominant architecture for large-scale distributed applications, powering everything from social media platforms to e-commerce systems. Yet despite this ubiquity, the complexity of these systems and the challenges of observing them remain poorly understood. The term "microservices" describes not a single design point but a vast space of configurations: different communication protocols, topologies, languages, and operational practices. Meanwhile, distributed tracing, the primary tool for understanding how requests flow through these systems, produces data that is routinely incomplete. Trace records are dropped during overloads, fields are missing, and the resulting traces may misrepresent the request workflows they are meant to capture.

This dissertation provides both a characterization of microservice complexity and techniques for observing these systems despite imperfect data. We first conduct a qualitative study combining interviews with 12 industry practitioners and analysis of six open-source benchmarks, revealing significant gaps between simplified academic testbeds and the heterogeneous, evolving systems found in production.

We then present two large-scale quantitative analyses. At Meta, we characterize 22 months of topology data and one day of request workflows, revealing over 12 million service instances, continuous churn, and high workflow variability that challenges assumptions about predictable system behavior. At Alibaba, we analyze two public trace datasets and discover that 85–99% of traces contain errors due to data loss and instrumentation failures. We develop Casper, a reconstruction tool that recovers traces 1.14–2.71$\times$ larger than naive methods.

Finally, we introduce bridges, a distributed tracing primitive that preserves trace characteristics despite data loss. Bridges propagate lightweight breadcrumbs that enable reconstruction of trace topology and ordering when intermediate data is lost. We present three bridge types with different fidelity/overhead trade-offs and demonstrate a prototype implementation in OpenTelemetry with minimal overhead.

Together, these contributions help practitioners and researchers understand the true complexity of modern distributed systems and provide tools for reliable observability even under realistic, lossy conditions.

To Dedication, without whom this would not have been dedicated.

Acknowledgments

Acknowledgments acknowledgments acknowledgments acknowledgments. Acknowledgments acknowledgments, acknowledgments acknowledgments acknowledgments acknowledgments acknowledgments acknowledgments. Acknowledgments acknowledgments acknowledgments acknowledgments: acknowledgments, acknowledgments acknowledgments, acknowledgments acknowledgments, and acknowledgments.

Acknowledgments acknowledgments acknowledgments acknowledgments acknowledgments. Acknowledgments acknowledgments acknowledgments acknowledgments acknowledgments acknowledgments acknowledgments (acknowledgments acknowledgments) acknowledgments. Acknowledgments acknowledgments acknowledgments acknowledgments acknowledgments acknowledgments acknowledgments acknowledgments acknowledgments acknowledgments. Acknowledgments acknowledgments.

Acknowledgments acknowledgments acknowledgments acknowledgments acknowledgments acknowledgments.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Over the past decade, microservices have become the dominant architecture for building large-scale distributed applications. Companies like Meta, Alibaba, Google, and Uber process billions of requests daily through interconnected networks of lightweight services, each responsible for a specific business functionality [1, 2, 3]. This architectural style enables development teams to work independently, deploy changes rapidly, and scale individual components based on demand. The shift from monolithic applications to microservices has been so pervasive that it now underpins a wide range of applications from social media feeds and search engines to e-commerce platforms and ride-sharing apps.

Despite this widespread adoption, microservices remain poorly understood outside the organizations that build them. The term *microservices* describes not a single design point, but an abstraction covering a vast space of possible configurations: different communication protocols, topologies, programming languages, deployment strategies, and operational practices. What constitutes a *service* varies between organizations. How services communicate, synchronously or asynchronously, via REST or RPC, differs based on use case. The number of services in a deployment can range from a handful to tens of thousands. This heterogeneity means that insights from one organization's microservices may not generalize to another's, and assumptions baked into research tools may not hold in practice.

Unlike static software artifacts, microservice architectures are living systems whose behavior is defined by runtime interactions. Distributed tracing is the primary tool for observing how services coordinate to fulfill user requests.Tracing captures graphs of request workflows as they traverse through the system. Traces reveal which services participate in processing a request, how long each service takes, and the dependencies between services on behalf of processing a request. They are the foundation for debugging, performance analysis, anomaly detection, and automated management tools like auto-scalers. Yet distributed tracing infrastructures are not infallible. They are treated as second-class citizens compared to revenue-generating functionality. The result is that real-world trace data is routinely incomplete. Trace records are dropped, fields are missing, and the traces that remain may not accurately represent the request workflows they are meant to capture.

This dissertation argues that meaningful research on microservice systems requires addressing both problems: we must accurately characterize how microservices behave in the real world, and we must develop observability techniques that remain useful even when the underlying data is incomplete. Through qualitative studies with industry practitioners, quantitative analyses of production systems and traces at Meta and Alibaba, and the design of new tracing primitives, this dissertation provides both the characterization and the tools needed to ground microservice research in reality.

1.1 Thesis Statement

In particular, this thesis states that:

Microservice characteristics vary widely across organizations, within single deployments over time, and even execution-to-execution, making observability critical. The observability data itself is abundant, but unreliable. This dissertation addresses both problems: through qualitative interviews, hyper-scale system analyses at Meta and Alibaba, and the design of bridges, we demonstrate that (1) academic testbeds diverge significantly from production microservices, (2) real-world traces suffer from critical data loss, and (3) lightweight ancestry data can preserve trace characteristics despite inevitable data loss.

1.2 Realistic Observability for Microservices

These challenges, fundamental variability and unreliable observability, motivate this research grounded in three complementary efforts. First, we must understand how microservices are actually built and operated in practice, moving beyond assumptions derived from small-scale testbeds. Second, we must quantify the characteristics of production microservices at scale, including how they change over time and how their observability data fails. Third, we must develop techniques that remain useful even when observability data is incomplete. This dissertation addresses each of these challenges through qualitative studies, hyper-scale system analyses, and the design of new tracing primitives.

1.2.1 Qualitatively exploring microservice characteristics

Academic research on microservices relies heavily on open-source testbeds such as DeathStarBench [4], TrainTicket [5], and TeaStore [6]. These testbeds provide reproducible environments for evaluating research including auto-scaling tools, debugging tools, and performance optimizations. However, it is unclear whether insights derived from these testbeds generalize to production microservices, which may differ in scale, complexity, and operational practices.

To bridge this gap, we conducted a qualitative study, published in the Journal of Systems Research [7], combining semi-structured interviews with 12 industry practitioners and a systematic analysis of six widely-used open-source testbeds. Our goal was to identify the design axes along which microservices vary and to assess whether existing testbeds adequately cover this design space.

We found significant divergence between testbeds and production systems. Practitioners reported using mixtures of synchronous and asynchronous communication, multiple programming languages, and evolving service boundaries driven by organizational needs. In contrast, most testbeds use a single communication protocol, are written in one or two languages, and have small, static topologies. Critically, practitioners emphasized the importance of

*evolvability,*the ability to add, remove, and version services over time, which no testbed adequately supports. These findings suggest that research validated only on existing testbeds may not transfer to production environments.

1.2.2 Quantitatively exploring hyper-scale microservice characteristics

To complement our practitioner study, we conducted two large-scale quantitative analyses: one characterizing Meta's microservice topology and request workflows, and another assessing the quality of Alibaba's publicly-released trace datasets.

Meta's Topology and Request Workflows (Section 4): In work published at USENIX ATC'23 [3], we present the first public characterization of Meta's microservice architecture, analyzing 22 months of topology data and one day of request workflow traces. Meta's topology contains over 12 million service instances and 180,000 communication edges between services. The topology evolves organically: thousands of services are created and deprecated each month. We also discovered that some software entities are *ill-fitting* to the microservices abstraction: they use service identifiers in non-standard ways that obscure their true complexity, such as machine-learning inference platforms that generate unique service IDs for each tenant.

Request workflows exhibit high variability even within a single day. The same parent service may call different sets of children across executions. This variability undermines tools that assume workflows are predictable from limited samples. Our findings quantify the scale and dynamism inherent to production microservices.

Alibaba's Trace Data Quality (Section 5): Alibaba has released two large-scale trace datasets, 12 hours of data from 2021 and 13 days from 2022, that have been widely used by the research community [8, 9]. However, no prior work has independently verified the quality of these datasets.

In work published at ACM/SPEC ICPE'24 [10], we conducted the first such analysis and found frequent inconsistencies. 99.48% of traces in the 2021 dataset and 85.77% of traces in the 2022 dataset violate at least one ofAlibaba's stated specifications for creating traces from the dataset. Inconsistencies include missing rows, contradictory field values, and duplicate records. We traced these errors to two root causes: data loss (records dropped by the tracing infrastructure) and context-propagation errors (services failing to correctly increment call identifiers). These inconsistencies cause naive trace reconstruction approaches, which assume the correct specifications, to produce smaller, disconnected graphs that misrepresent the original request workflows.

To address this, we developed *Casper*, a trace reconstruction tool that exploits hidden redundancies in the datasets to recover from data loss and context-propagation errors. Casper produces traces that are 1.14–2.71× larger in size, 1.22–1.32× deeper, and 1.08–2.02× wider than naive methods. It increases the fraction of fully-connected traces from 58% to 84% in the 2021 dataset.

1.2.3 Robust and Accurate Observability for Microservices

Alibaba's trace data model encodes topological redundancies that enable some error correction, but other tracing models do not have this attribute. Our analysis of Alibaba, Uber, and Meta traces reveals that 5–40% of traces contain *holes*: contiguous sequences of missing records where the tracing infrastructure dropped data. Holes may occur when observability is most valuable: during overloads and failures, when services shed non-critical work like trace collection to preserve revenue-generating functionality.

Existing tracing infrastructures, including the industry-standard OpenTelemetry, cannot reconnect trace fragments across holes. Traces are constructed by linking each recorded operation to the operation that invoked it; when intermediate records are lost, the operations they invoked become disconnected from the rest of the trace. Tools that assume complete traces like anomaly detectors, auto-scalers, and critical-path analyzers may produce incorrect results when applied to fragmented data.

Existing tracing infrastructures, including the industry-standard OpenTelemetry, cannot reconnect trace fragments across holes. Current designs only preserve local connectivity—each record knows what directly preceded it, not the broader context of the request. When records are lost, this local information is insufficient to piece the trace back together. Tools that assume complete traces like anomaly detectors, auto-scalers, and critical-path analyzers may produce incorrect results when applied to fragmented data.

We introduce *bridges*, a new distributed tracing primitive that preserves important trace characteristics despite data loss. Bridges work by propagating lightweight *breadcrumbs* containing compact metadata needed to reconnect trace fragments across holes. We present three bridge types with different fidelity and overhead trade-offs. The *Structural Bridge* preserves both call-graph topology and ordering relationships between sibling calls, enabling critical-path extraction. The *Call-Graph Bridge* preserves topology but not ordering, reducing overhead while still preserving accurate trace topology. The *Path Bridge* preserves only connectivity, minimizing overhead for use cases that require only fragment linkage.

Bridges are designed for practical deployment: they require no changes to application business logic, only the use of our modified OpenTelemetry SDK. Our implementation, *Manchac*, demonstrates that bridges can be added to existing instrumented applications with minimal overhead. Evaluation on DeathStarBench Social Network shows that bridges recover connectivity for traces with holes while adding minimal latency overhead. This work is ongoing, with submission planned for Spring 2026.

1.3 Contributions

Overall, this thesis contains the following contributions:

Proposed Ideas/Systems:

- A framework for comparing microservice testbeds against production characteristics
- First public characterization of Meta's microservice topology and request workflow dynamics.
- Analysis methodologies for identifying inconsistencies in Alibaba's trace data set as well as methods for accurately correcting errors.
- *Bridges*: a new distributed tracing primitive that preserves trace characteristics across data loss.

Artifacts:

- Analysis scripts and error correction code for Alibaba's trace data set.
- Corrected sample of Alibaba traces.
- Bridges implementation in OpenTelemetry and Blueprint.

1.4 Dissertation Organization

The remainder of this dissertation is organized as follows:

- Chapter 2 provides the background necessary to understand the research questions and contributions of this dissertation.
- Chapter 3 qualitatively explores microservice characteristics via a user study and compares the themes to commonly used open-sourced microservice testbeds highlighting limitations in the testbeds' design choices compared to the design space found in industry.
- Chapter 4 quantitatively explores Meta's microservice architecture at scale. We describe the topology in detail on one day and how it changes over a 22-month period, then examine request workflow characteristics including variability and predictability.
- Chapter 5 analyzes Alibaba's system through the lens of their publicly-released trace datasets. We identify pervasive data quality issues and present CASPER, a reconstruction tool that accurately repairs errors.
- Chapter 6 develops the *Bridges* primitive, which adds minimal redundant data to traces that allow for recovery of important trace characteristics when data is lost. We build a prototype and show initial evaluation results.

Chapter 2

Relevant Background

2.1 Microservice architectures

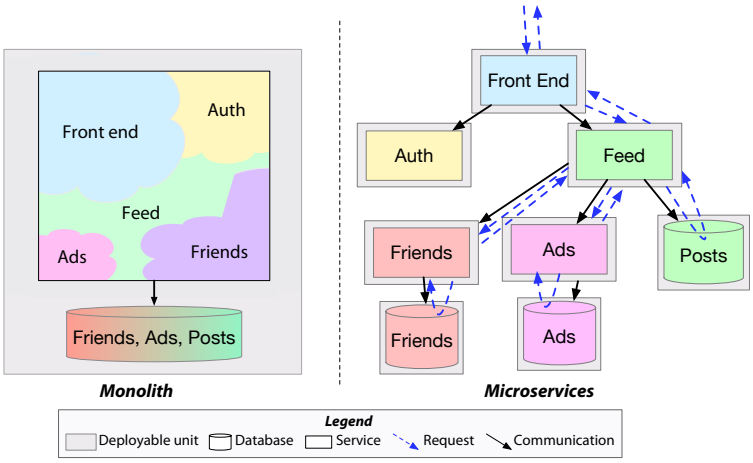


Figure 2.1: **Monolith *vs.* Microservices** A monolith is a single deployable unit, as illustrated on the left. A microservice architecture, shown on the right, is composed of multiple deployable units that communicate with each other. A request workflow is shown on the microservice application.

Microservices is an architectural style in which a large-scale application is decomposed into individual services that work together to achieve a business goal. Figure 2.1 contrasts two architectural styles for a social network application. In a monolithic architecture, all functionalities are built into a single deployment unit that interfaces with centralized databases. In a microservice architecture, the same application is realized through independent services—Authentication, Feed, Friends, and Ads—each responsible for a specific part of the business domain. Services may have their own storage mechanisms rather than depending on a shared database [11, 12].

Services and endpoints: Services are units of software with well-defined API interfaces called *endpoints*. Each service satisfies a specific *business use case* (e.g., caching a photo feed), though there is room for interpretation in defining the scope of a use case. Additionally, software that pre-dates the microservice architecture may serve multiple business use cases, but be deployed as a single service. Both services and endpoints are named by respective services' developers. Services expose their functionality via publicly accessible endpoints and may be replicated for scalability; replicas are called *instances*. Additionally, services can be shared across application boundaries. At large organizations like Meta, microservice topologies are highly dynamic, with services frequently created and deprecated [3, 2, 1].

Stateful and stateless services: Services can be stateful or stateless [1]. Stateful services, such as databases,

persist state for callers. Stateless services, such as frontends, call other services and integrate their results without maintaining persistent state. A variety of programming languages are used to write services depending on fit for the business use case and organizational conventions.

2.2 Request workflows

When a user interacts with a microservice application, their request is processed by a subset of services in the topology. Services that receive external user requests or originate internal ones are called *root services*. Root services call other services' endpoints to perform the desired work, and those services may in turn call others. The sequence of calls made on behalf of a single request forms a *request workflow*, which can be represented as a directed acyclic graph (DAG) where nodes are service executions and edges indicate caller/callee relationships. figure 2.1 shows an example request workflow, loading a user's feed, in blue.

Parent/child relationships: Within a workflow, a service that calls another is the *parent*, and the called service is the *child*. A parent may call multiple children, either sequentially or concurrently. Sequential calls occur when parents are single-threaded or when there are data dependencies between calls. For example, if an authentication token must be returned from one service before being passed to others. Concurrent calls occur when children can execute independently.

Critical paths and latent work: The *critical path* of a request workflow is the longest chain of sequential dependencies that determines overall latency. When children are called concurrently, only the slowest child contributes to the critical path. When called sequentially, all children contribute. Children may also perform *latent work* after replying to the parent which does not affect the parent's response time but is part of the workflow. For example, asynchronous replication or garbage collection.

Service communication: Services communicate via various communication methods including remote procedure calls (RPCs), HTTP requests, or via a message queue. Communication can be *two-way*, where the caller blocks or waits until receiving a response, or *one-way*, where the caller continues without waiting. Two-way communication is typical for request/response patterns like RPCs, while one-way communication is used for latent operations like logging or notifications.

2.3 Distributed tracing

Distributed tracing is the primary observability mechanism for understanding request workflows in microservice architectures. Traces are directed acyclic graphs (DAGs) that capture how requests are processed by a system, revealing which services participate, how long each takes, and where errors occur.

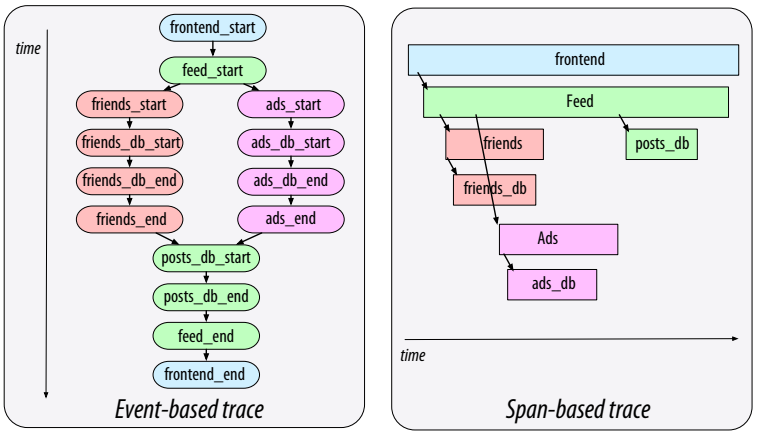


Figure 2.2: **Event vs Span traces** shows the same trace represented via event-based and span-based trace data models.

Trace data models: There are two main tracing data models: event-based and span-based. Figure 2.2 shows an example of a single request workflow represented as both an event-based and span-based trace. In event-based traces, nodes represent discrete events annotated with a name and timestamp, connected by *happens-before* relationships [13]. Fan-outs capture work that can be done in parallel, while fan-ins represent synchronization points where a service waits for multiple downstream calls to complete before continuing. In span-based traces, nodes are *spans*, intervals of work such as function executions, annotated with start time, duration, and optional key-value pairs describing the work performed. Spans are connected by *caller/callee* (parent/child) relationships. In practice, most industrial systems use span-based tracing.

Tracing infrastructure: Figure 2.3 illustrates a typical tracing infrastructure. The leading open-source standard is OpenTelemetry [14], which includes SDKs, tracing agents co-located with services, and centralized collectors. The SDK instruments application code to create spans and propagate *baggage*, small amounts of metadata including a unique *trace ID* and *parent span ID*, along request paths. Each span is assigned a unique *span ID* and annotated with its trace ID and a parent span ID to preserve caller/callee relationships.

Span records are cached locally by agents co-located with each service, then sent asynchronously to a central collector. The collector joins records with the same trace ID to reconstruct complete traces. Trace records typically include span name, timestamps, error codes, and may include additional metadata.

Context propagation: Building traces requires propagating context along request paths. Most tracing infrastructures use *one-way context propagation*, where context flows only on the forward (call) path. Some systems, primarily academic ones, use *two-way context propagation*, where context also flows on the response path. Two-way propagation enables richer relationships such as fork/join patterns and precise ordering between sibling calls but adds complexity. For example, the critical path in a 2-way context propagation model (preserving happens-before relationships) is defined by finding the longest path from start to end of a request workflow while the critical path

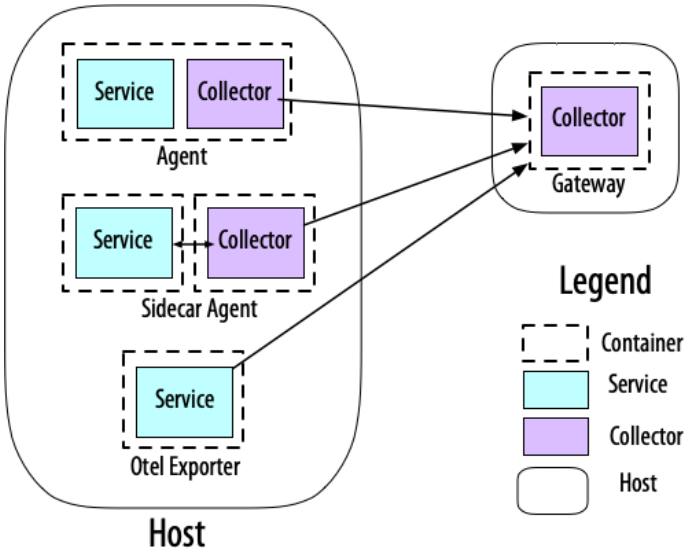


Figure 2.3: **Collector deployment landscape.**

of a one-way context propagation trace (preserving parent/child relationships) is an approximation of the critical path, defined by following the last ending child algorithm [**?**]. In practice, one-way propagation is dominant despite the additional expressiveness of two-way approaches.

Sampling: At scale, tracing every request is prohibitively expensive. Organizations use sampling strategies to reduce overhead while preserving representative data. *Head-based sampling* decides whether to trace a request at its origin, before execution begins. *Tail-based sampling* defers the decision until after execution, allowing selection based on observed characteristics like latency or error codes. Both approaches involve trade-offs between overhead, storage cost, and the completeness of collected data.

Chapter 3

Qualitatively

Exploring Microservice Characteristics

3.1 Overview

Industrial microservice architectures vary so wildly in their characteristics, such as size or level of modularity, that comparing systems is difficult and often leads to confusion and misinterpretation. In contrast, the academic testbeds used to conduct microservices research employ a very constrained set of design choices. This lack of systemization in these key design choices when developing microservice architectures has led to uncertainty over how to use experiments from testbeds to inform practical deployments, and indeed whether this should be done at all. We conduct semi-structured interviews with industry participants to understand the representativeness of existing testbed design choices. Surprising results included the presence of cycles in industry deployments, as well as a lack of clarity about the presence of hierarchies. We then systematize the possible design choices we learned about from the interviews, and identify important mismatches between our interview results and testbeds' designs that will inform future, more representative testbeds.

3.2 Background

The term "microservices" is credited to a 2011 presentation by Netflix [15, 16]. In the early days, the large business case handled by an organization was combined into a single executable and deployable entity, which is referred to as *monolithic architecture*. Though the functionality of an application grew linearly with increasing business case, each user's access of different features were non-uniform [17]. To circumvent disadvantages of monolithic applications like single-point failure, multiple organizations decomposed their applications into various functionalities but retained a common communication bus to facilitate communications between different components[18]. This is called Service Oriented Architecture (SOA). Microservices evolved from SOA, where the common communication bus was replaced by an API call from one service to another.

Early academic research in microservices focused on the impact of domain characteristics when migrating from a monolithic to a microservice paradigm [19, 20, 21, 22, 23, 24, 25, 26]. These extensive studies produced insights on how multiple organizations handled various design choices such as service boundaries, cost of re-architecture and

	DSB - SN	DSB - HR	DSB - MR	TrainTicket	BookInfo	μSuite	TeaStore
Communication							
Protocol	Apache Thrift	gRPC	Apache Thrift	REST	REST	gRPC	REST
Style	-	-	-	Async	-	-	-
Languages Used	C/C++	Go	C/C++	Java	Node, RoR, Java, Python	C++	Java
Topology							
Number of services	26	17	30	68	4	3	5
Dependency Structure	Hierarchy[1]	Hierarchy	Hierarchy	Hierarchy[2]	Hierarchy	Hierarchy	Hierarchy
Evolvability							
Versioning Support	No	No	No	No	Yes	No	No
Perf. & Correctness							
Distributed Tracing	Jaeger	Jaeger	Jaeger	Jaeger	Jaeger\Zipkin	None	Jaeger
Testing Practices	U,L	U,L	U,L	U,L	L	L	E2E,L
Security							
Security Practices	TLS	TLS	TLS	None	TLS via Istio	None	TLS via Istio

Table 3.1: **Design choices of microservice testbeds.** The table shows axes by which existing microservice testbeds vary. It also shows testbeds choices for these axes. *In some* indicates that databases are included within some services, but are not separate services. *One single* indicates that all services share a single database, which is itself an independent service. U=Unit Testing, L=Load Testing, E2E=End-to-End.

infrastructure, and monitoring tools, along with challenges faced by developers in terms of implementation of large scale distributed systems. In a similar vein, multiple projects [27, 28, 29, 30] have examined how to decompose existing monolithic architecture into microservices. All of these works focused on static analysis, which was based on functionality, rather than the dynamic traffic experienced by these systems.

More recently, microservice research has shifted focus from migration to a more holistic analysis of microservices, ranging from surveys, to testbeds, to tools to better understand the trade-offs of practical microservice design [31, 32, 33, 34, 35, 36, 37, 38, 39, 19, 40, 41, 42]. Of particular note, Wang *et al.* [43] produced a large survey on post adoption problems in microservices, with questions focusing on the benefits and pitfalls of maintaining large scale microservice deployments. We extend the areas explored in published literature and compare it with open source microserivce testbeds.

3.2.1 Microservice Testbeds

Following the growth of microservices in industry, the academic world has embraced the concept by building multiple applications for multiple use-cases using microservice architectures. In our work, we refer to the overall group of applications as testbeds, and to an individual use-case as an application. For this work, we only selected the testbeds whose code is Open Source, and available to be deployed on any platform of choice. These open-source testbeds provide transparency and reproducibility to microservice research, and enable multiple follow-up research projects.

3.2.1.0.1 DeathStarBench Gan *et al.* [44] released this testbed suite in 2019 to explore the impact of microservices across cloud systems, hardware, and application designs. This testbed suite has been the most widely used by researchers. The suite is built based on the 5 core principles: Representativeness, End-to-End Operation, Heterogeneity, Modularity and Reconfigurability. These principles were adopted to make the testbed appropriate for evaluating multiple tools, methods and practices associated with microservices. Each application has a front end webpage from which users can send requests to an API gateway which routes it to appropriate services and compiles the result as an HTML page. DeathStarBench consists seven applications as testbeds: Social Network, Movie Review, Ecommerce, Banking System, Swarm Cloud, Swarm Edge and Hotel Reservation. In this paper, we only looked into three of those: Social Network (DSB-SN), Hotel Review (DSB-HR) and Movie Review (DSB-MR), because their code is Open Source and has ample documentation for deployment, testing and usage.

3.2.1.0.2 TrainTicket Zhou *et al.* [**?**] released this testbed in 2018 to capture long request chains of microservice applications. To build this testbed, the developers interviewed 16 participants from the industry, asking about common industry practices. The major motivation to build TrainTicket was the limitation of existing testbeds' small size and the need for a more representative testbed. The authors specifically asked about various bugs that occur in microservice applications and replicated them in this testbed. The authors subsequently used this testbed to test these bugs or faults and developed debugging strategies. There are multiple requests that can be sent to the application to login, to display train schedules, to reserve tickets and to do any other typical functionalities for a ticket booking application. The requests enter a gateway and are routed to the appropriate services based on the request, with results compiled and sent as responses to the HTML frontend.

3.2.1.0.3 BookInfo BookInfo [45] was developed as part of Istio Service Mesh [46] to demonstrate the capabilities of deploying microservice applications using Istio. This testbed is an application that displays information for a book, similar to a single catalog entry of an online book store. It consists of 4 services: Product page, Details, Reviews and Rating. The requests are sent to the product page, which gets the necessary information from the other 3 services, aggregates the results, and shows it in an HTML page.

3.2.1.0.4 μSuite Sriraman *et al.* [47] released this testbed in 2018 to evaluate operating system and network overheads faced by Online Data-Intensive (OLDI) microservices. It contains 4 different applications – HDSearch, a content based search engine for images; Router, a replication-based protocol router to scale key-value stores; SetAlgebra, an application to perform Set Algebra operations on Document Retrieval; and Recommend, a user based item recommendation system to predict user rating. The applications were built to understand the impact of microservice applications on the system calls, and underlying hardware. This testbed was geared towards Online Data Intensive applications, which handles processing of huge amounts of data using complex algorithms. All the applications have an interface which allows for the users to run them on a large scale dataset and record the observations.

3.2.1.0.5 TeaStore Kistowski *et al.* [6] released this testbed in 2018 to test the performance characteristics of microservice applications. The testbed consists of 5 services: WebUI, Auth, Persistence, Recommender and Image Provider along with a Registry Service which communicates with all the other services. The Registry Service acts as the entry point for requests and requires each service to register their presence with this service. The testbed can also be used with any workload generation framework, and has been tested for Performance Modeling, Cloud Resource Management and Energy Efficiency analysis. This modular design enables researchers to add or remove services to the testbed and customize them for specific use cases. The application caters to multiple requests for working with a typical e-commerce application such as login, listing products, ordering products. The requests enter using the WebUI service, which sends a request to the registry service that routes the requests to appropriate services, aggregates the result, and displays the result as HTML webpage.

Overall, while there are multiple testbeds available, most academic papers used DeathStarBench, specifically DSB-SN, which is the Social Network Service [48, 49, 50, 51, 52, 53, 54, 55]. The next most widely used testbed is TrainTicket [52, 37, 36, **?**], and the other testbeds are used less commonly in the academic research community.

3.2.2 Testbeds' Design Choices

When building these testbeds developers make choices about various individual aspects of the application. In this section, we explore the choices made by the original developers of the testbeds and illustrate the various options used to build them. We look at both the literature and the codebase of the testbeds for various design choices, in matters of conflict we pick the option illustrated in the codebase as it receives constant updates from the developers and larger community. An overview of the design choices and the options adopted by the various testbeds are shown in Table 3.1.

3.2.2.1 Communication

Communication choices refer to the required methods and languages used for building each of the services, as well as for interfacing between the different services. They form the bedrock on which the application is built, as

they enable the information passing between the services to execute requests. We analyze the testbeds to identify the communication *Protocol* between two internal microservices, as it impacts the performance of applications [56, 57, 58]. We also identify whether the *Style* of communication is synchronous or asynchronous, and further analyze the testbeds to identify the *Programming Languages* used for implementation, as microservice architectures provide the flexibility of using multiple languages.

3.2.2.1.1 Protocol TrainTicket, BookInfo, and TeaStore use REST APIs for communicating between different services to complete a request, and also for communication between the webpage and initial service. DSB-SN and DSB-MR use Apache Thrift for communication between the services, but has a REST API for communication between the Web Interface and the gateway service. DSB-HR and all applications in μSuite use gRPC for communication between the services. DSB-HR uses a REST API for communicating between the webpage and gateway service whereas μSuite makes use of gRPC for the same purpose.

3.2.2.1.2 Style BookInfo, TeaStore, DSB-HR, and DSB-MR only have synchronous communication channels between the various services and do not use any data pipelines or task queues for coordinating asynchronous requests in their applications. TrainTicket has both synchronous and asynchronous REST communication methods between the services across the application. DSB-SN uses synchronous Thrift channels for communication between the services, but has a RabbitMQ task queue that is used for asynchronous processing of some requests such as compiling the Home Timeline service for a user after they create a new post. μSuite has both synchronous and asynchronous gRPC communication channels for each of the applications built separately with no overlap between each other.

3.2.2.1.3 Languages Used All the services in TeaStore and μSuite are built using only one language: Java and C++ respectively. The services that process business logic in DSB-SN and DSB-HR are built using C++. Lua is used for processing the incoming request and compiling the final result sent to users, Python is used to perform unit tests and for smaller scripts that are used to setup the testbed. DSB-MR and all the applications in μSuite are written using Golang. BookInfo consists of 4 services, each of which has been written in a different language: Python, Java, Ruby and Javascript (Node.js). The services in TrainTicket are also written in 4 languages: Java, Python, Javascript, and Golang. All testbeds except μSuite offer a user interface written using HTML, CSS, and JS.

3.2.2.2 Topology

Topology relates to the overall structure of the application including the communication channels between the services. We look at the ways in which different testbeds have arranged the services to fulfill requests for a particular application. We look at the number of services and the dependency structure of an application. The number of services is counted as the total number of containers (services + storage) that needs to be deployed for the application to fulfill

all its requests[3]. In testbeds where containers are not used, we went by the individual deployments. The topology is often represented as a Dependency Diagram as shown in Figure 2.1, where the nodes represent services and an edge from Service A to Service B means Service A is dependent on Service B to complete a request. Request call graphs, determined by the overall topology, yield important insights, as shown in Alibaba's large-scale traces of microservices [59]. We analyze the testbeds to identify that the overall dependency structure of the microservices was *hierarchical*, where the request entered an API gateway as the first service and the storage layer was the last accessed service.

3.2.2.2.1 Number of Services μSuite [60] has 4 distinct applications, each of which have only 3 distinct services [4]. BookInfo has 4 distinct services each deployed as a container within Istio Service Mesh [61]. TeaStore has 5 distinct services with a Registry Service that keeps track of the total number of services in the application [62]. TrainTicket [63] has 68 services including the databases which are deployed as separate containers. DSB-SN[64] has 26 individual containers including the databases and caches, DSB-HR [65] has 17 individual containers including the databases and caches, and DSB-MR [66] has 30 individual containers including the databases and caches.

3.2.2.2.2 Dependency Structure μSuite was built under the assumption that the OLDI microservices are hierarchical in nature, where the application is structured as front end, mid-tier, and leaf microservices. BookInfo is also structured in a hierarchical structure where the nodes at the end are storage services such as MongoDB. TrainTicket doesn't follow a strictly hierarchical structure, as the database isn't the last layer accessed for some of the requests. DSB-SN, DSB-HR and DSB-MR are strictly hierarchical as the requests entering the API gateway go through each service before accessing the database towards the end of the request chain, from where it is directly returned to the user. DSB-SN has a non-hierarchical component where the Home-Timeline gets compiled asynchronously when a user creates a new post. TeaStore has a hierarchical dependency when processing requests, however every newly deployed service calls the Registry Service to register itself.

3.2.2.3 Evolvability

As the application becomes larger, the architecture changes based on the various modifications that each individual service undergoes. We analysed the testbeds to check if they had already incorporated this design axes in their application. We also looked at the support for versioning in the testbeds to gauge the support for multiple versions of the same service [67, 68]. For example, as shown in Figure 3.1, Service A and B are dependent on Service C to fulfill their request and they use the API `/api/service_c`. If Service C is modified to accommodate newer features, or code optimizations, these changes might not be adapted by Service A or B at the same time. Thus,

[3]This count was retrieved on 29 th January, 2022

[4]We derive this number from the installation script provided by the authors in their code [60]

Service A will be using the older version (v1) and Service B will have moved to the newer version (v2). This would require Service C to run 2 instances with different versions to support all their dependencies.

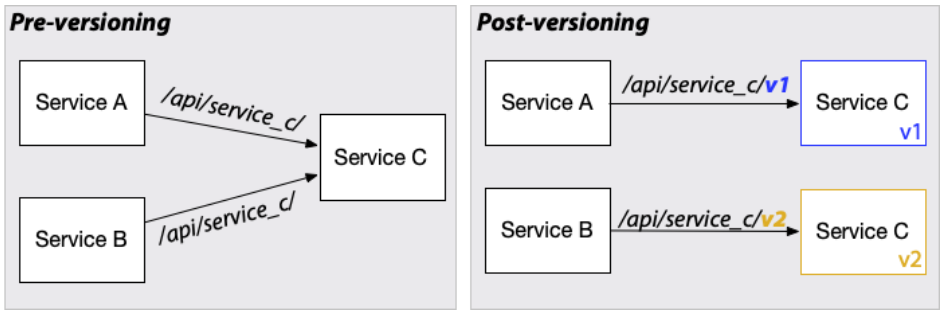


Figure 3.1: **The Versioning Problem:** one approach to maintaining multiple versions of a service is by using versioned APIs.

3.2.2.3.1 Versioning Support Only BookInfo provides multiple versions of a service in its testbed. The Reviews Service comes in 3 different versions where 2 of the versions access the Ratings Service to display the ratings on the webpage. Other testbeds do not explicitly provide multiple versions of the services but have extensible APIs that the users can program to deploy multiple versions of a service.

3.2.2.4 Performance & Correctness

Understanding and analyzing the performance of microservices is integral to designing microservices. We analyze the testbeds to identify the different *Distributed Tracing* tools adopted by the testbeds for analyzing the performance of each service in the request chain [69, 70].

3.2.2.4.1 Distributed Tracing Except μSuite all the other testbeds offer Distributed Tracing built into the testbed. These testbeds use a framework built on OpenTracing principles, typically with Jaeger as the default option. They instrument each of the applications with various tracepoints built into each of the services to track the time spent processing each request. Though it doesn't use distributed tracing, μSuite uses eBPF to trace various points of the system to get the number of system calls that were being utilized to run various applications in the testbed.

3.2.2.4.2 Testing Practices Except μSuite all the other services have unit testing built into the repository which can be used to test the individual services for correctness. TeaStore also has an end-to-end testing module that interfaces with the WebUI service to mimic a user clicking the UI. Load testing can be performed on all the testbeds except μSuite using wrk2[71] since they use HTTP for receiving requests. μSuite has an inbuilt load generator in the codebase that can be used for generating higher request loads to test the application.

3.2.2.5 Security

3.2.2.5.1 Security Practices DSB-SN, DSB-HR, and DSB-MR have a Transport Layer Security built-in between the services which helps in encrypting communication between the services. TeaStore and BookInfo were deployed using Istio Service Mesh which comes with built-in encryption channels that can be enabled by the developer when deploying the application. MicroSuite and TrainTicket do not provide communication encryption between the services.

3.3 Methodology

We conducted semi-structured interviews with industry participants to *1)* better understand the designs of industrial microservices and *2)* understand how these designs contrast with those of available testbeds. Our IRB-approved study follows the procedure shown in Figure 3.2.

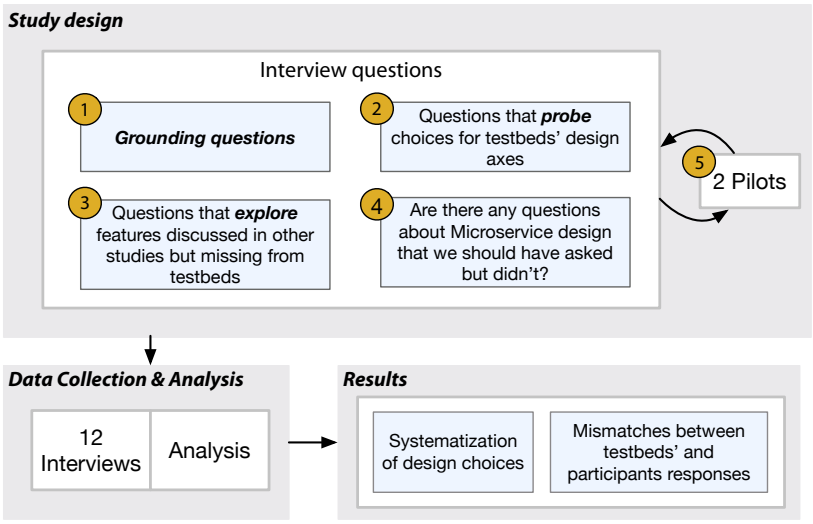


Figure 3.2: **Methodology:** The interview process starts with study design, followed by data collection & analysis, and ends with our results.

3.3.1 Recruiting Participants

We recruited participants from different backgrounds, aiming to collect various perspectives of microservice design choices. (See Appendix **??** for the demographics questions we asked.) We recruited participants by: *1)* reaching out to industry practitioners and *2)* advertising our research study on social media platforms (Twitter, Reddit, and Facebook). After the first few participants were recruited, we used snowball sampling [72, 43] to recruit additional participants. We recruited fourteen participants in total, including the two pilot studies (see below).

Table 3.2 shows demographics of the participants we recruited for our interviews. The table shows that out of the

ID	Skill level	YoE	Sectors worked	Current role
P1	Advanced	10	Government	Full Cycle
P2	Intermediate	3	Finance, Tech, Government, Consulting, Education	Full Cycle
P3	Advanced	5	Tech	Full Cycle
P4	Beginner	1	Tech, Research	Design, Testing
P5	Advanced	5	Finance, Tech, Education	Full Cycle except Deployment
P6	Advanced	4	Tech	Full Cycle except Deployment
P7	Advanced	10	Academia, Tech	Full Cycle
P8	Intermediate	3.5	Tech	Design, Testing, Implementation
P9	Intermediate	2	Tech	Full Cycle except Scaling
P10	Advanced	7	Tech	Deployment
P11	Advanced	7	Tech, Government, Consulting	Full Cycle
P12	Intermediate	2	Tech	Full Cycle except Scaling

Table 3.2: **Participant Demographics** Each participant, which can be identified by their *ID*, has their self reported *skill level*, years of experience *YoE* with microservices, *sectors worked* in with respect to microservices, and *current role*. Full Cycle covers all the five aspects of microservices: design, testing, scaling, deployment, implementation.

twelve participants, seven assess their skill level with microservices as advanced-level, four as intermediate-level, and one as beginner-level. On average, they have five years of experience working with microservices. Sectors that the interviewees work in include government, consulting, education, finance and research labs. 9 of the 12 interviewees work on all aspects of microservices, (defined as design, testing, scaling, deployment and implementation). The remaining 3 work only on a smaller subset of those aspects.

3.3.2 Creating Interview Questions

We created 32 interview questions designed to increase the authors' understanding of industrial microservice architectures and to contrast microservice testbeds with them (see Appendix **??**). The questions span four categories, described below.

(1) **Grounding questions**: These questions ask participants to define microservices and state their advantages and disadvantages. We use these questions to determine whether participants exhibit a common understanding of microservices, and whether this understanding agrees with that described in previous literature [44, 37, 6, 25, 43, 39, 19, 40, 41, 42].

(2) **Probing questions** These questions probe whether design elements present in microservice testbeds accurately reflect or are narrower than those in industrial microservices. For example, Table 3.1 shows that all microservice testbeds exhibit a hierarchical topology where leaves are infrastructure services. So, we asked whether microservice topologies can be non-hierarchical. We asked similar questions about tooling. For example, only one

out of the seven testbeds include versioning support. So, we asked whether industrial microservices at participants' organizations include versioning support.

(3) **Exploratory questions**: These questions center around microservice design features discussed in the literature [43, 73, 31, 74]), but completely missing from the testbeds we consider. For example, cyclic dependencies within requests—*i.e.*,, service A calling service B which then calls A again—occur in Alibaba traces [59], but are not present in any of the testbeds we analyzed. This mismatch led us to investigate if request-level cyclic dependencies occur in participants' organizations. Similarly, the testbeds do not make statements about application-level or per-service SLAs (*i.e.*,, the minimum performance or availability guaranteed to caller over a set time period [75, 76, 77]). So, we asked questions about whether microservice architectures within participants' organizations include SLAs.

(4) **Completeness check question**: We ended each interview by asking if there is anything about microservice design that we should have asked, but did not. This question helped us gain confidence in the systematization we report on in Section 3.4. (Though, we cannot guarantee comprehensiveness.)

(5) **Pilot studies**: We conducted two pilot studies before the first interview. We refined the interview questions based on the results of these pilots.

3.3.3 Interviews & Data Analysis

Our hour-long interviews consisted of a 5-10 minute introduction, followed by the questions. Participants were told they could skip answering questions (*e.g.*,, due to NDAs). We encouraged participants to respond to our questions directly and also to think-aloud about their answers. We asked clarifying questions in cases where participants' responses seemed unclear and moved on to the next question if we were unable to obtain a clear answer in a set time period. At times, we probed participants with additional (unscripted) questions to obtain additional insights.

For data analysis, three of the co-authors analyzed participants' responses together. We used the labels below to categorize responses. We additionally identified themes in the interview answers and extracted quotes about them.

1. *Unable to interpret*: The three co-authors' could not come to a consensus on the interpretation
2. *Unsure*: Interviewees did not know the answer
3. *Yes*: for a yes-or-no question
4. *No*: for a yes-or-no question

We report only on participants who provided answers and whose answers we can interpret (hence the denominators for participants' responses in Section 3.4 may not always be 12).

3.3.4 Systematization & Mismatches

Systematization: We used the responses to our questions to expand the testbed design axis table presented in Table 3.1 and create Table 3.3. New rows either correspond to *1)* exploratory questions about microservice design that elicited strong participant support or *2)* design elements a majority participants verbalized while thinking out loud. Columns correspond to specific technologies or methods participants discussed for the corresponding row.

Mismatches: We compared the results of our expanded design axis table to the table specifically about testbeds, in order to identify cases where the testbeds could provide additional support.

3.4 Results

Table 3.3 describes the design space for microservices based on the testbeds and interview results. The rows are grouped into high-level design categories including Communication, Topology, Service Reuse, Evolvability, Performance & Correctness, and Other. Within each category, there are specific design axes along with the range of responses from participants and specific examples, when applicable. For example, the communication category includes specific axes for protocol, style of communication, and languages used.

In the following sections, we discuss each row of Table 3.3. We first state the number of participants who provided responses that were interpretable. We then state the high-level results, which are applicable to all of our participants. We also present specific granular breakdowns for each result where applicable. Following these statistics, we provide quotes from the interviews, referencing participants by their ID in Table 3.2.

3.4.1 Grounding questions

Participants' responses were similar to results in existing user studies [43, 32] and other academic literature [44, 37, 6]. In describing what microservices are, 7 out of 12 participants identified them as independently deployable units and 3 participants explicitly mentioned that applications are split into microservices by different business domains. Almost all participants noted the ease of deployment, testing, and iterating on services as being benefits of microservices. On the other hand, a monolith was described by most participants as a single deployable unit with all of its business logic in one place. Participants noted that monoliths have many downfalls, such as their inability to scale granularly, having a tight coupling of components, and being a single point of failure.

While participants agreed on common benefits like isolated deployment and failures, they disagreed on the challenges caused by using microservices. Concerns range from high-level views, such as difficulty with seeing the big picture of the whole application, to more specific ones like extra work (*e.g.*, getting data from a database) caused by strict boundaries and backwards compatibility (*e.g.*, the versioning problem). Regarding how shared libraries

Design Axes	Range of Responses	Examples
Communication		
Protocol	HTTP, RPC, both	gRPC, REST, Apache Thrift
Style	Synchronous, Asynchronous, Both	-
Languages Used	Multiple - Restricted, Multiple - Unrestricted, One	Java, Python, C\C++, Go
Topology		
Number of Services	Varies	8-30, 50-100, 1000+
Dependency Structure	Hierarchical, Non-Hierarchical	-
Cycles	Yes, No	-
Service Boundaries	Business Use Case, Cost, Single Team Ownership Distinct Functionality, Performance, Security	-
Service Reuse		
Within an Application	Yes, No	-
Across Applications	Yes, No	-
Storage	Shared, Dedicated, Both	-
Evolvability		
Versioning Support	Yes, No	Versioned API, Explicit Support (UDDI), Proxy
Perf. & Correctness		
SLAs for Microservices	Yes - Applications, Yes - Applications and Services, No	-
Distributed Tracing	Yes, No	Jaeger, Zipkin, Homespun
Testing Practices	Unit, Integration, End-to-End, Load, CI\CD	-
Security		
Security Practices	Granular Control, Communication Encryption Attack Surface Awareness	-

Table 3.3: **Design Space for Microservice Architectures** These design axes were identified through the practitioner interviews. Rows in the table, which are specific design axes, are grouped by design category. Each design axis has the *range of responses* from the interviews as well as specific *examples* of specific design choices mentioned by the interviewees.

and microservices are distinct, most participants were unsure of a true distinction, while some tied microservices to stateful entities and shared libraries to stateless entities.

3.4.2 Communication

3.4.2.0.1 Protocol We have 11 interpretable responses for the communication protocols used at participants' organizations. 5 of the total 11 responses included HTTP, and 6 responses had a combination of both HTTP and RPCs. No participants use only RPCs for communication. For these communication protocols, participants shared specific mechanisms including REST APIs (6/11) and gRPC (3/6).

Of the eleven participants that mentioned using HTTP as a communication protocol, three of them mentioned

using standard HTTP without mentioning REST specifically. Two participants shared that any communication protocol can be used, beyond HTTP and RPCs, in appropriate scenarios.

Participants expressed differing opinions on which communication protocol is best suited for microservice applications, with P2 saying "in the real world [use] REST... if your team needs RPC you're probably doing some sort of cutting edge problem" since "the overhead for using REST is relatively negligible to RPC," while others, such as P9, felt more drawn to RPCs: "we use both [HTTP and RPC], but generally we would prefer to use RPC."

3.4.2.0.2 Style We have 5 interpretable responses for the communication styles used at participants' organizations. 3 of the 5 participants with interpretable responses suggested that their organizations have a mixture of both synchronous and asynchronous communication styles in their services, while the remaining 2 participants only mentioned synchronous forms of communication.

Out of the three participants that use both forms of communication, P5 warned of the dangers of poor design combined with only synchronous communication saying "you certainly don't want a scenario where somebody has to make multiple calls to multiple services and all those calls are synchronous in a way that is hazardous and... I think folks are mindful of this when they make broad designs. I think this starts to break down when folks are trying to make nuanced updates within." P3 also noted that one benefit of asynchronous communication is that "[dependencies are] more dotted lines than solid lines right, they're not strictly depended on this." Additionally, P1 pointed out that "[logging] is completely asynchronous," indicating a specific use case for asynchronous communication.

3.4.2.0.3 Languages Used We have interpretable responses for all 12 participants regarding the languages used at their organization. Participants' responses included 3 restricted to using only one language, 4 using multiple languages with restrictions on which ones could be used, and 5 using multiple languages with no restrictions.

All three participants that only use one language at their organization are restricted to using Java. P1 attributed this to their hiring pool: "...management will typically look at what's cheaper in the general market. Which technical skill sets are readily available in case someone leaves and they need to replace [them] and so on."

Out of the four participants who used a restricted set of languages (more than one), P8 shared that using a small set of languages is due to "shared libraries. If you have very good shared libraries that make things super easy in one language and if you were to switch to another language, even if you like writing in that language, there's almost no... Look, at the end of the day, the differences between languages are not [great enough] to be able to throw away a lot of shared libraries that you would otherwise be able to use."

Out of the five participants who have unrestricted language choices, P2 explained that "some of these [services] were forced to use a [new] language because the library is only available for this language."

Out of the nine participants that use multiple languages, six use three to five languages in their applications,

two use more than eight languages, and one did not know the number, saying " I'd go to Stack Overflow and [ask] how many languages exist?" (P4). Table 3.3 shows the most commonly used languages among our participants' organizations: Java, Python, C\C++, and Go.

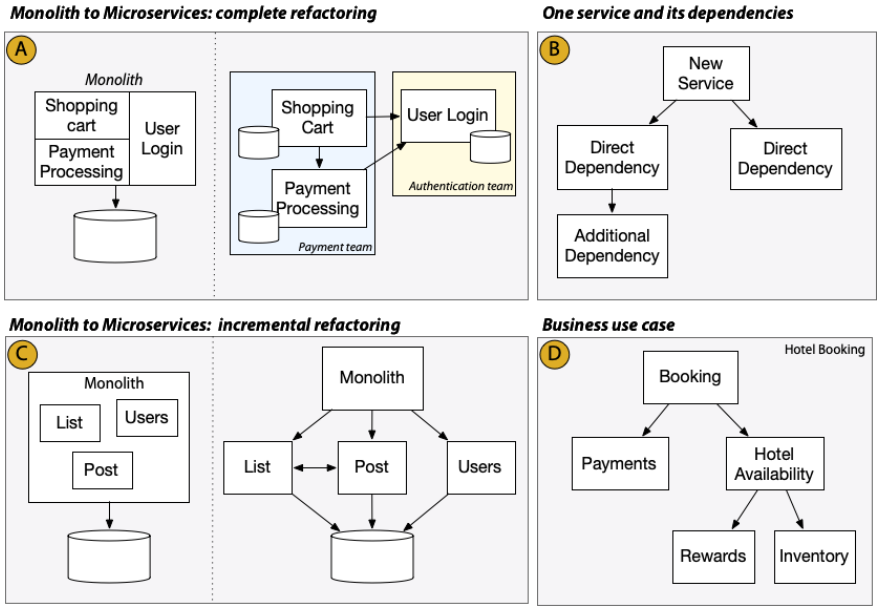


Figure 3.3: **Topology Approaches** For the most part, participants used one of four approaches when asked to draw a microservice dependency diagram that would be used to explain microservices to a novice. Note that (C) represents a hybrid deployment retaining some monolithic characteristics.

3.4.3 Topology

We asked participants to draw a Service Dependency Diagram to explain microservices for a novice entering the field. This gave us a sense of the important characteristics of microservices that participants think about most prominently. 3 of the 12 participants drew two different diagrams, giving us 15 total diagrams. We present these results in Figure 3.3 showing the most common approaches taken by participants.

The first common approach was to draw a monolith then completely refactor it into a microservice architecture (1/15, (A)). The second approach was similar, starting with a monolith and pulling out specific bits of functionality into microservices. This incremental refactoring approach resulted in a monolith connected to a set of microservices (3/15, (C)). The third approach was to take one service and expand the architecture by considering its dependencies (4/15, (B)). The final and most popular approach was to consider a business use case, listing all services needed to accomplish the task, then connecting the dependencies (6/15, (D)). The single other approach, which is not included in the figure, was centered on container orchestration (P4).

3.4.3.0.1 Number of Services We have 12 interpretable responses for the number of services in the applications managed by participants' organizations. As shown in Table 3.3, the number of services ranged from 8-30 services (3/12), 50-100 services (5/12), and over 1,000 services (1/12). The responsibility of development and maintenance of these services is shared across multiple teams at the organizations. (3/12) participants were unsure of the number of services at their organizations.

Of the 3 participants that were unsure, P7 explained that "I can't [estimate the number of services] because it depends how you divide. For example, I have some services that run multiple copies of themselves as different clusters with slightly different configurations. Are those different services or not?... Not only could I not even tell you the count of them, I can't tell you who calls what, because it might depend on the call and it could change day to day."

3.4.3.0.2 Dependency Structure We have 10 interpretable results for participants' experiences with microservice dependency structures. The responses consisted of hierarchical (2/10), non-hierarchical (6/10), and unsure or no strong stance either way (2/10).

Most participants rejected the notion that microservice dependency structures are strictly hierarchical. Recall that a hierarchical topology is one where services can be organized as a tree, where the top level services are an API gateway or load balancer and the leaves are storage. Participants often initially said yes, but then changed their minds and thought of counterexamples. For example, P11 explained "now that I'm evaluating microservices and and I'm recognizing that the services should be completely independent, there's no reason that they should always follow that paradigm... I'm coming to an answer, no, it is not always the case." Participants provided different reasons for non-hierarchical topologies. For example, both P7 and P8 described non-root entry points: "I guess the way I think about it [is], where does work originate. And it is perfectly valid for it to originate from outside the microservices or from inside the microservice architecture, so I think it can go both ways"(P8).

Of the two participants that agreed microservice dependency structures are strictly hierarchical, both attributed this belief to only having experiences with hierarchical topologies. For example, P9 said "all the ones I've seen have been that way I guess. I can't rule out the there may be some other reason to architect [it] another way, but yeah I would agree [that microservice dependency diagrams are strictly hierarchical]."

3.4.3.0.3 Cycles We have 9 interpretable responses for cyclic dependencies in microservice dependency diagrams. Most (6/9) participants agreed that there can be cyclic dependencies between services in microservice based applications, while the remaining (3/9) participants were unsure.

Six participants expressed that cyclic dependencies should be avoided when possible. For example, P4 explained "generally speaking... you kind of want to avoid cycles just for keeping your designers sane so I don't think it's a good idea." In a similar vein, P6 shared "whether or not a microservice, you know, service one could hit service two and

service two could hit service one again? Yeah, I would say that it's... I would consider that an Anti pattern, but not- Like there still might be a good reason to do it, but I would consider that generally is like a code smell or a stink."

Three participants did not warn against cyclic dependencies. P8, for example, explained that "[valid cyclic dependencies] depend on how you divide your microservices. [For] many microservices, it just makes sense to have them contain both the front end and kind of the business logic like back end code. And sometimes, microservices borrow things. When a microservice is holding back end and front end code, you could imagine service A calling service B to get, you know, some nice back end calculation done. But then maybe service A actually has a cool widget that service B wants to display... You could be super rigid- like this microservice just has this one really cool widget. Well, then you would never have any cycles, because each [microservice] only does one thing. But that's not practical like things are going to host front end components. And the links between the front end components and back end logic are not always as hierarchical."

3.4.3.0.4 Service Boundaries We have 9 interpretable responses for service boundaries. Participants listed many different ways to create service boundaries: by business use case (2/9), by single team owner(4/9), in ways that optimize performance (3/9), in ways that reduce cost (2/9), and by distinct functionality (2/9). (Participants provided multiple answers, so our tallies add up to more than nine.)

The most frequent answer among the participants was setting service boundaries to have single team ownership. P2 warns of "the pain of having an improperly scoped business domain where multiple teams are trying to compete basically for the same bit of business logic. Make only one team [responsible] for that logic even if... multiple have to co-parent, one needs to be accountable." P3 explains, when reflecting on refactoring one microservice into a user and enterprise service, "we had two different teams that were going to be focusing on different things and iterating on those things very, very quickly."

P4 discusses how overheads could change due to service boundaries: "If I'm able to do everything internally by just sharing memory buffers or just shooting little message queues around, that's one thing. If I suddenly have to communicate through a bunch of HTTP [requests] or sockets [due to refactoring my service], am I adding additional overhead in there that may be degrading my performance in a meaningful way?" In addition, P4 weighs the security costs of having more, smaller services: "suddenly let's say we decompose [one service] into four things. Each one of these might have a different attack surface that we need to reexamine. Is it worth the cost of looking into that?"

3.4.4 Service Reuse

3.4.4.0.1 Within an Application We have 4 interpretable responses about service reuse within a single application. All 4 participants indicated a significant amount of service reuse within an application.

P2 explained "you always have a few [services] that everybody is dependent on." In addition to this, P12

shared that microservices could be reused within an application, specifically for different endpoints. They added that when microservices are reused within an application, they "don't think the same request would go to the same microservice twice, that seems like bad engineering to me. You should be able to do everything you're supposed to do on the first time around."

3.4.4.0.2 Across Applications We have 9 interpretable responses about services being used in multiple applications. (8/9) said some of their services were shared across applications while (1/9) said none of their services were reused.

Of the participants who said services were shared across applications, P2 said it's "pretty common" and P12 said "that is why we made microservices." In a similar vein, P7 explained "that's almost always, yeah. With the exception of maybe the very front end of them". P8's organization has "some core services that are used by all applications." For example, they mentioned "the authentication service is used by all."

P4, an industry researcher, explained "we have a specific service which we have actually containerized to test these things out and we are looking at potentially having multiple applications ping it." As for how many dependencies this shared service would have, P4 said, "it's going to depend on what we're trying to research. In this case, since we are doing some research on scalability, we will eventually very deliberately go through and see how many different things can we connect to it before it falls over sort of thing."

As for the participant whose organization does not reuse service across applications, P11 shared that "I don't see that. It's just one application and it's just a collection of microservices bounded by that application context that mirrors the silo that the application is built in." When asked if the same functionality was required for two different applications, they shared that "they would generally be making a new microservice to fill" the need.

3.4.4.0.3 Storage We have 10 interpretable responses about database reuse. The response included dedicated database per service (3/10), shared databases (5/10), and a combination of both (2/10).

Of the three participants that have only dedicated databases for their services, P8 shared "the one thing I can say is that [our] core services will have their own devoted data store so like authentication, [has an] authentication database." They could not share information about their application specific services' databases. P11, a consultant, said dedicated databases "is what I'm seeing most often, yes."

Of the five the participants with databases used by multiple services, P3 said "ideally, they don't. In practice, they absolutely did." Not all participants felt as though databases shouldn't be shared. For example, P2 explained "they always share! Every time, they always share." P6 said "my previous company definitely reused databases. Microservices and teams might have their own tables within that database, but that database was still the same."

Finally, P5 shared that "we have a legacy database. In fact, every one of our customers has its own database. That's necessary for compliance reasons."

Of the two participants whose organizations have a combination of dedicated and shared storage. P9 initially explained "each [service] I'm aware of uses a dedicated storage," but later added "there is a microservice that can be used for storage that I guess, in a sense, is a way storage can be shared."

3.4.5 Evolvability

3.4.5.0.1 Versioning Support We have 11 interpretable responses about how participants approach adding new versions of a microservice. 6/11 participants have some sort of versioning support in place while the remaining 5/11 did not. As shown in Table 3.3, the methods of versioning support used by the participants include versioned APIs (2/6), explicit support like UDDI (1/6)[78] and a proxy (1/6). The remaining 2/6 participants with versioning support did not provide a specific mechanism.

Of the six participants that have a mechanism in place for adding new versions of services, P1 shared "there's things like UDDI that help with versioning, but we typically don't depend on that. We will literally just publish a new endpoint." P7 explained using a proxy for versioning, where a copy of a small amount of production traffic would be routed to the new version instead of the old one and the results of the two versions would be compared. P3 shared the preference to "translate internally. Right, so a request can still come to the old [version], but you're just using the new code."

The two participants that did not provide a specific mechanism for versioning explained that they use Blue/Green Deployment for verifying that new versions should be shipped to production.

Of the five participants that did not have a mechanism in place for versioning, P9's approach to adding new versions is to "deploy into a different version of the cluster. That's how I test my services- manually configure the route headers to contact this test cluster." P11, shared that at the companies they consult at, they "for lack of a better option they're simply coding and hoping that it will say the same."

Two of the participants shared the challenges with versioning. P2 warned "that's the problem with microservices that you're coming to... that no matter what [with] microservices you get into a dependency hell. The biggest thing I can say is [please] version your API. If you're going to use an API, version it and have some sort of agreement for how many old versions you want to maintain." P12 explained that at their previous company, they deemed this "the versioning problem... Your change in your one domain, when you're updating the microservice, has to be reflected company wide on anything that depends on it or utilizes it, and so I mean there are ways to handle this which is like, you know bend over backwards, for the sake of backwards compatibility." They explained their process for versioning as "whenever a microservice gets changed, try to determine through... regular expression code search where all of the references in the code base to that particular stream of characters were but that's not enough. So what you

then have to do is you'd have to actually grunge through the abstract syntax tree of each Python program in order to determine the parameters that were given [and] the types of those parameters." They ended this discussion with "I've left that job and I continue to [think] about it on a near weekly basis because it's such an interesting problem."

3.4.6 Performance & Correctness

3.4.6.0.1 SLAs for Microservices We have 11 interpretable responses about SLAs with respect to microservice based applications and microservices themselves. 8/11 have SLAs with the remaining 3/11 not having SLAs. Of the 8 participants that have SLAs, 7/8 have SLAs for entire applications and 3/8 have SLAs for individual microservices.

Of the eight participants that have SLAs, P6 explained "we had SLAs with respect to the entire product's behavior and the product was composed of the microservices. So as a unit the microservices had an SLA which was like, we wanted four nines reliability like 99.99% uptime. But that was considering the product as a unit not as the microservice. We did, internal to the company, have individual targets where... it was just part of like your performance review as a team." Plus, P1 shared "we have SLAs for everything, [including individual microservices]."

The remaining three participants expressed varying sentiments on why their organization did not have SLAs. For example, P3 explained a challenge of supporting SLAs: "there [were] a lot of fights about it. It was one of those things I wish we did. But I think before you can have those... like there were things we were missing to tell what service level you're actually offering. And before you can have agreements we have to know how to measure if you're actually hitting those agreements or not. That was a rather consistent argument between the engineering teams and the infrastructure teams." On the other hand, P5 explained "we're not known as high availability and we're not.... Nothing is transactional or urging in that particular way" as the reason for not needing SLAs at their organization.

3.4.6.0.2 Distributed Tracing We have interpretable responses for all 12 participants on whether they use distributed tracing. 1/12 was unfamiliar with distributed tracing, 8/12 did not use distributed tracing, and 3/12 did use distributed tracing. As shown in Table 3.3, of the participants that use distributed tracing, one uses Zipkin, one uses Jaeger, and one uses a homespun tracing framework. Of the participants that do not use distributed tracing, 2/8 want to and 2/8 understand the need for tracing.

Of the three participants that use distributed tracing, P7 explained "we use Zipkin... we rely on the features that are enabled by it so it shows things like service dependencies, we use [it] for capacity planning, we use it for debugging. If we want to know why there is a performance problem, my team doesn't do a lot of this right now because there hasn't been a lot of pressure on that, but, other teams do look at this and they're like 'Why is [there] a performance problem?' and they'll look at the traces and be like 'oh yeah this call is taking three times as long as you'd expect'." P10 shared that "we have multi-tenancy environments meaning we have multiple customers, multiple people accessing

the same services." P10 also shared how they use the trace data to "[get] management in place, in other words, when you step into a cluster, by default- it's a free for all... everything [can] talk to everything. What you really want to start doing is...basically [build] highways/roadways inside the cluster and [define] those roadways... and we actually apply policy for our applications so that... We know that this namespace and this "pod" and the services [are] talking to the parts and services it's exposed to, and nothing else. You want to prohibit that kind of anomalous activity."

Of the nine participants that don't use distributed tracing, P1 shared "we're not that far yet." P2, a consultant, explained that "normally by the time that's really a problem, fortunately I'm out of there... I'm more involved in the early few months of work. If you're into that level of debugging, you're normally months in or years and something's gone really wrong somewhere and you're trying to figure out who broke it." Finally, P3 explained "there are some places that it got set up [but] I didn't have too much experience with it. That was one of those [things] if we had invested more time into it, we would have gotten more out of it. We just never really invested the time."

3.4.6.0.3 Testing Practices We have 11 interpretable responses about testing practices with respect to microservices. The most common tests are unit tests (9/11), integration tests (5/11), end-to-end tests (4/11), load testing (4/11), and using a CI\CD pipeline (3/11). (Participants provided multiple answers, so our tallies add up to more than eleven.)

Participants listed a wide variety of testing types and practices in addition to the ones listed above including smoke tests, static code analysis, chaos testing, user acceptance testing, and so on. Even with an abundance of available testing methods, some participants, including P9, "stick to testing the individual functionality of the microservice." Other participants aim to expand their testing practices as their company grows. For example, P11, a consultant, shared "blue green canary deployments... those are things that we talked about but it doesn't happen there- [the companies are] not mature enough to do that."

Two participants expressed dissatisfaction with the testing practices at their companies. For example, P3 shared "where we could, we would do load testing, but I have yet to see a place that does that particularly well. It's really hard to mimic [production] load in any sort of staging environment. It's really hard to mimic [production] data in any sort of staging environment." P5 explained that they use "end-to-end [testing], but for the product broadly, [the tests are] incredibly flimsy. And they're hard to write, so a lot of our microservices that we think are tested are not tested." In addition, P2 shared another testing challenge: "What do I do when I'm dependent on another thing changing? That's a great question and [I] still do not have a good answer for that."

As a result of the challenges of testing microservice based applications, some participants shared a different mindset about testing. For example, P3 explained, "at some point, in some places we cared less about testing before the thing went out and more being able to very quickly un-break it when it does break."

3.4.7 Security

3.4.7.0.1 Security Practices We have 11 interpretable responses about security practices with respect to microservices. Three themes emerged among the responses: exercising granular control over security (4/11), encrypting communication (4/11), and having awareness of your attack surfaces (3/11).

Since microservices have well defined endpoints and boundaries, it is possible to have granular control over the security of each endpoint. P5 explains "you can have really clear granular control, about which [services] can communicate with which other [services]" and what the service is allowed to do. For example, "service A might have some users that are only authorized for certain GET calls. And other services [are] authorized perhaps maybe to write certain things, but it should not be able to ask questions of that thing. And then yet another service has the right to write to a queue that that service will eventually pick up and do something with that, but doesn't otherwise give any knowledge of what's there." P7 echoed this sentiment by saying "you may have different trust boundaries on the different services." P3 explained that security efforts can be focused on certain aspects of a system, asking "do we need to care about this? In many cases, no. Admitting logs to a log server like if you're not logging sensitive information, who cares? Sending billing data back and forth, like, I care a lot. So it depends on what bits you care about."

In addition to focusing security efforts, participants pointed out that communication between services should be secure. For example, P7 said "you have to deal with the network, so your network has to be secure." P1 agreed that "with microservices you're typically having to encrypt and secure the communication between services themselves... given the chatty-ness of them and the fact that they're typically communicating over REST APIs, you need to secure all of that. It's handled typically, at least in my world, using sidecar injections and containers and so on. " Not all participants agreed who should be responsible for communication encryption. P2 shared "the reality is that nobody cares about security, they push it off to the... so I'm a security nerd. [But,] developers don't care about security... If your subnet can truly be trusted, [it's] not an issue. But if you can't and run into issues with eavesdropping, this is something where having a service mesh can help basically encrypting those connections." P6 explained "I would say some organizations can probably get away with less strict security practices, where if you're internal to their network, they don't have to be as careful. They're not encrypting the traffic. They're not using TLS because they're assuming that everything's locked down, all the hardware [it's] running on is locked down and no one else can access it. And if you're in their network, you're in their network, so it doesn't really matter."

With microservices, the number of externally available end points can have an impact on security. P8 shared "if all your microservices are publicly exposed to the Internet, someone can enter that topology from any node" which would make penetration testing more difficult as well as tracking down malicious actors. In addition, P4 explained "your attack surfaces [with microservices] look fundamentally different on some level."

Participants shared that microservices can simplify security. For example, P9 said "it's a lot easier to audit

	DSB - SN	DSB - HR	DSB - MR	TrainTicket	BookInfo	MSuite	TeaStore
Communication							
Style	Both	Sync	Sync	Async	Sync	Both	Sync
Topology							
Cycles	No	No	No	No	No	No	No
Service Boundaries	BUC, STO	BUC, STO	BUC, STO	DF	DF	Three Tiers	Performance
Service Reuse							
Within an Application	Yes	Yes	Yes	Yes	Yes	No	No
Across Applications	No	No	No	No	No	No	No
Storage	In Some	In Some	In Some	In Some	In Some	None	Dedicated
Perf. & Correctness							
SLA for Microservices	Supported	Supported	Supported	Supported	Supported	Supported	Supported

Table 3.4: **Additional design axes for microservice testbeds** These new design axes were discovered after conducting practitioner interviews. *In some* indicates that databases are included within some services, but is not a separate services. *Dedicated* indicates that a separate service interfaces with all the databases, and exposes an API for other services. BUC=Business Use Case, STO=Single Team Ownership, Three Tiers meant each application is just three tiers deep.

your security concerns in a microservice architecture, just because you have to define each of your individual dependencies." Similarly, P11 said "from a microservices standpoint you would typically expect a higher level of scrutiny of the code, because you have better visibility, things are more discrete."

3.5 Recommendations and Analysis

The interview results we present in Section 3.4 illustrate that there are a series of gulfs between the assumptions under which testbeds (§ 3.2.1) are designed and the expectations and needs of users and architects in production-level microservice deployments. Following the key design considerations outlined in Table 3.3, we analyze the discrepancy between testbeds and the systems they claim to represent, as well as providing guidance for creating a more representative microservice testbed. We expand on the findings of newer design axes in Table 3.4.

3.5.1 Communication

We compare the design decisions that developers in industry make regarding communication protocol, style, or language with the choices made by the testbeds. Overall, the testbeds encompass the wide range of options used

by industry practitioners, but diverge in the finer design aspects of communication channels in microservices.

3.5.1.1 Protocol

The first decision that developers need to make regards the way services communicate with each other. Typically, the entry point to a service will be using a REST API, as most microservices applications are accessed using a browser or mobile application. REST APIs, however, lack the performance benefits offered by specific RPC frameworks such as *gRPC* or *Apache Thrift* [56, 57, 58]. Using an RPC framework requires the application to be more rigidly defined, reducing its resilience and adaptability.

Academic Testbeds: TrainTicket, BookInfo and TeaStore makes use of REST and DSB-SN, DSB-HR use Apache Thrift, while μSuite, along with DSB-MR, use gRPC. No testbed uses more than one communication protocol.

Interview Summary: Even though all participants agree that their application contains both REST or RPC in appropriate scenarios, 7/11 participants leaned towards REST for its robustness and ease of implementation. Participants also indicate using a mixture of both these protocols, as some parts of the application might be more latency sensitive.

Recommendation: There is a need for testbeds that have a mixture of REST and RPC protocol(s) within the same application to replicate a section of the use cases seen in the industry. Choosing a communication protocol has significant effects on latency, resource utilization, and other characteristics of the application [79, 80]. Thus, an application with a mixture of these protocols would help us measure and mitigate effects of various protocols on resource utilization, latency, etc.

3.5.1.2 Style

The style of communication impacts the performance of the application, with asynchronous services having higher throughput than synchronous services [81, 82]. This increased performance comes with more complex faults, as the requests might arrive out of order or get dropped in transit.

Academic Testbeds: The major communication channels between the services in testbeds are synchronous in nature, with some testbeds having some services which process information asynchronously. DSB-HR, DSB-MR, TeaStore, and BookInfo do not use any asynchronous communication in their architecture.

DSB-SN is the only DSB application with an asynchronous component that helps in populating the WRITE-HOME-TIMELINE service, which constructs the home timeline and stores in a cache. This makes use of message queues for the asynchronous calls between services. TrainTicket is the only testbed that contains both asynchronous REST calls and Message Queues. μSuite applications have two variants; synchronous and asynchronous as two separate applications in the codebase.

Interview Summary: Participant studies show two ways to implement asynchronous communication: asynchronous requests between two services and using a message queue. The overall findings can be summarized by a quote from Participant 5: "You certainly don't want a scenario where somebody has to make multiple calls to multiple services and all those calls are synchronous in a way that is hazardous and... I think folks are mindful of this when they make broad designs, I think this starts to break down when folks are trying to make nuanced updates within." Asynchronous updates can also be a part of design choices arising from the requests originating from within an application, a design choice we discovered during our conversations with practitioners.

Recommendations: There is a gap in understanding the impact on Asynchronous RPC calls in a synchronous setting. This presents an opportunity for expanding the existing testbeds to include asychronity, particularly in handling message queues. There is also a need for understanding the impact of periodic internal requests on the performance and resource utilization of the application.

3.5.1.3 Majority Languages

We track the programming language used across testbeds and compare them to the languages our participants reported for their applications.

Academic Testbeds: To make this comparison, we only look at the language that was used to write core application logic. DSB-SN and DSB-MR are largely written in C++ as the core language, with Python being used for testing the RPC channel, Lua for interfacing between external requests and internal applications, and C for workload generation. DSB-HR is completely written in Golang. TrainTicket and BookInfo use 4 languages, Java, Javascript, and Python being the common languages, with TrainTicket opting for Golang and BookInfo choosing Ruby as the other language. In TrainTicket, the majority of the services are written using Java, whereas BookInfo has 4 services, each of which is written in a different language. All the applications in μSuite are written in C++ and do not use any other language. TeaStore is completely written using Java, with Javascript used in parts for integration purposes.

Interview Summary: While 75% the participants indicated using multiple languages for their applications, half stuck with a few core languages for the majority of their services, and experimented with other languages based on specific needs. The major reason for using a limited set of languages was to leverage the power of core libraries which are available for those particular languages. Java, Python and C# are the most commonly used languages of development among our participants' organizations.

Recommendations: Overall, we find that the diversity of languages is similar across the industry and academic testbeds. While some testbeds, such as TrainTicket, work with multiple languages using REST, there is a need for benchmarking polyglot applications that make use of RPC communication mechanisms, as there is fluctuation in performance and resource utilization between implementations of RPC mechanisms in different languages [83].

This will help application developers make better decisions on the choice of language used to build a specific part of an application.

3.5.2 Topology

Topology has a profound impact on individual requests' response times and the overall latency of the application. We compare the structure of microservice testbeds with microservice characteristics observed in actual implementations. Overall, the large, intricate connectivity of microservice topologies (colloquially referred to as "Death Star graphs" due to a resemblance to certain space stations) is not reflected in the capabilities of the benchmarks. Topology also has impacts beyond application, where it can dictate the way in which software engineering teams are set up as well [84, 85] [5].

3.5.2.1 Number of Services

The number of services in a microservice-based application is based on the business domain and goals of the organization. Participants had varying definitions for what constituted a service; however, for the testbeds, we counted a single service to be a container that is deployed in production.

Academic Testbeds: μSuite, BookInfo and TeaStore have fewer than 10 services. DSB-SN, DSB-MR, DSB-HR have 26, 30 and 17 services respectively, with scalability tests performed using multiple deployments of the existing services. TrainTicket has 68 services in their testbed, and in the original work [37], they mention this not being representative of the scale at which industry operates.

Interview Summary: Half of our participants' organizations had worked with more than 50 services in their architecture, with the services split between multiple teams which were responsible for development and maintenance of the services. One of the participants also shared that it was impossible to count the number of services in production, as the number was not static; it changed periodically due to new services being added, breaking down existing services to manage at least one load, deploying replicas of existing services, or deploying newer versions of existing services.

Recommendations: The number of services in testbeds do not represent the true scale of these applications. This is evident from our survey data, as well as published reports which state that typical microservice deployments include hundreds of services [37, 59, 86]. There is still no single testbed that mimics the scale of services in industry, thus presenting an opportunity for an industry scale testbed for performing scalability and complexity studies.

3.5.2.2 Dependency Structure & Cycles

Understanding and emulating the dependency structures that define microservice topology is critical to provisioning, tracing, and failure analysis. This is one of the areas of strongest mismatch between testbeds and actual use, particularly in the presence of cycles.

[5] This is referred to as "Conway's Law."

Academic Testbeds: All of the testbeds we studied follow a hierarchical topology, with requests originating from outside the system. DSB-SN, DSB-HR, DSB-MR and TrainTicket have multiple requests for testing different functionalities of the application, but the trace graph for each of these requests shows a hierarchical ordering of services. μSuite has only one type of request, which goes through the three tiers of the application before returning a result. BookInfo uses 3 versions of the REVIEWS service where the request might not reach ratings services in 1 version but is required in the other 2. TeaStore also has one request type originating externally, which goes through all the services in a linear manner before returning a response. Because of their strict hierarchical models, we assume no cycles can be present.

Interview Summary: Most of our participants reported that microservice architectures are not strictly hierarchical, where the root node might be an API gateway with storage layer in the leaf node and other application logic in between. They are more non-hierarchical, with some requests originating from within the system, and a few have cyclic dependencies as well [59]. The participants that assumed hierarchy noted that their assumptions were from lack of experience or exposure to non-hierarchical systems, indicating that the limited topology in academic microservice work may be actively limiting them.

Recommendations: Existing testbeds are universally hierarchical in request processing, which does not represent the majority of production systems we encountered. More accurate representation would enable researchers to study and develop tools for a broader variety of realistic dependency structures. Moreover, there is an opportunity for testbeds to include more flexibility in storage models, such that different caching configurations and privacy-preserving data placements are easier to analyse. Finally, a **key finding** of our study is that the presence of cycles in microservice architectures is a theme in industrial deployments (validated further in a recent study from Alibaba [59]), but completely absent in existing testbeds. Along with hierarchy, testbeds also need to have cyclic dependencies in order to study the effects of such dependencies on tools revolving around deployments, tracing, and scaling.

3.5.2.3 Service Boundaries

Given the modular nature of microservice architectures, there is a need for understanding the motivation behind creating these service boundaries. We compare the motivations behind creating such boundaries in industry and academic settings, and provide recommendations on the ways in which these gaps can be bridged.

Academic Testbeds: All the DeathStarBench applications have been demarcated using "Business Use Case" and encouraging "Single Team Ownership". TrainTicket and BookInfo have distinct functionality for each of the services in their architecture, whereas TeaStore services are conceived to maximise the performance of the system. In contrast to the industry practices, μSuite was built with three tiers as the basis for all microservice applications, a design choice that is different from the industry practitioners.

Interview Summary: While the industry practitioners provided various responses for splitting service boundaries,

the most common response was to split it based on Single Team Ownership, where each service is owned by a single team in accordance with Conway's Law [84]. They also talked about the dynamic aspect of microservices where a single service can be decomposed into multiple services based on variety of factors specific to organizations. New services can also be added due to expanding the feature set of a product. However, the caveat of spawning multiple new services is that this adds communication overhead placed on the system, with new network calls being made to various services.

Recommendations: Most of the existing testbeds are built as static communication graphs, but the industry practitioners, and also the literature [87, 59, 86, 88, 89], tend to look at microservices as dynamic entities. Since the testbeds are built with extensibility as a core design pillar, researchers can extend existing testbeds to accommodate for newer services. This can be used for comparing the performance and resource utilization of the application before and after the changes.

3.5.3 Service Reuse

Microservice architecture literature, and the testbeds derived therein, assume each service is built with loose coupling and high cohesion in order to maximise service sharing and minimizing duplicate code. We compare the extent of sharing of services between the industry implementations and academic testbeds.

3.5.3.1 Within an Application

Academic Testbeds: The testbeds are built with a principle of modularity, which is a core tenet of microservice architecture. Applications in DeathStarBench (DSB-SN, DSB-MR, DSB-HR) and TrainTicket have a modular design wherein a service can be accessed by other services based on the needs of each request. When looking at each request chain that emerges in the traces, there is little overlap between the different services used for processing different kinds of request.

Interview Summary: A third of the participants pointed out some level of sharing of existing services in their architectures, noting sharing as one of the major benefits of the microservice architectures. Sharing of services ranges from sharing key infrastructure services to large parts of application code.

Recommendations: Even though service sharing is portrayed in testbeds, the level of sharing does not entirely match practices in industry. This can be fixed by creating new features which would use the existing services as well as extending the current functionality of the testbeds.

3.5.3.2 Across Application

Academic Testbeds: Only DeathStarBench and μSuite have multiple applications which can be used for analyzing the sharing of services across applications. When looking at their traces and the codebase, there is no

overlap or reuse of services between their applications.

Interview Summary: The participants whose organizations had multiple applications indicated that they reuse services between different applications as well. The extent of this ranged from sharing parts of the application such as authentication to sharing critical infrastructure services such as logging.

Recommendations: Testbeds with multiple applications can be modified to share services among the different applications for reuse between multiple services. Since the various applications have different access patterns, this would help researchers study the effects of mixed application workloads on the performance and resource utilization of services.

3.5.3.3 Storage

Academic Testbeds: All the testbeds currently have the storage layer in their leaf nodes, or towards the end of the request chain. The testbeds, with the exception of μSuite applications, use a variety of persistent storage (both SQL (MySQL) and NoSQL (Mongo)) for storing the data. μSuite applications do not make use of any persistent storage, as the dataset to run the testbeds were stored as CSV files. DSB-SN, DSB-MR, DSB-HR, and μSuite use a caching layer of memcached or redis to store the transient results for faster access. TeaStore has a specific service which acts as as interface between the database and other services. This gives them the flexibility to swap out the database without the application being affected.

Interview Summary: From our interviews, we did not get a consensus on a single kind of criteria for placement of databases in Microservice architecture. Some organizations preferred having single database per microservice for ease of maintenance, while others preferred this design only for critical services such as authentication. Many participants preferred having shared databases, at least in non-critical parts of the application, with the exception of one participant who mentioned always sharing the databases.

Recommendations: While placement of storage is subject to the design and use case of the application, the testbeds do not have extensive sharing of databases with each other. The testbeds can be extended to explore the design paradigm of database sharing where multiple services access the same data store for retrieving information. This would be useful to explore, particularly in the context of privacy regulations such as GDPR [90, 91]. There is also literature that has explored the field of caches for microservices, for example placing caches based on workloads experienced by each service [92].

3.5.4 Evolvability

When evaluating the design in terms of production capabilities, we deployed each of the testbeds on machines using the instructions provided in their repositories.

3.5.4.1 Versioning Support

Academic Testbeds: Only BookInfo offers a single service with multiple versions which can be used for evaluating versioning support. Similar to Adding Services, other testbeds provide avenues by which a researcher could edit existing services and re-deploy as separate versions. TrainTicket, TeaStore and BookInfo use REST which can be easily extended by writing another version of a service in any language and modifying the request chain. The service can be deployed using Docker container and given a new REST API endpoint which is interfaced with other services. DSB-SN and DSB-MR use Apache Thrift, while DSB-HR and μSuite make use of gRPC as their communication protocol. Adding or removing versions of services is more complex in these cases as the underlying code-generation file needs to be modified with updated dependencies, then application code must be written for the newly generated service, which then must be deployed using Docker.

Interview Summary: The survey results indicate that managing versioning is a problem in active microservice deployments and that there is no consensus on how to address it. Some engineers deploy new versions as a separate service, and systematically fix the errors that occur because of these changes. Participants used existing methods and tools to alleviate the problems that arise when having multiple services are running concurrently.

Recommendations: To catalyze academic research into the versioning problem, we recommend that testbeds be extended to readily allow for multiple versions of the same service in order to help understand the effects on performance.

3.5.5 Performance Analysis Support

3.5.5.1 SLA for Services

SLAs are used for comparing the necessary metrics of a service to ensure a promised level of performance, and define a penalty if that level is not met.

Academic Testbeds: Existing testbeds can define SLAs, and resources can be allocated based on the traffic experienced by the service. SLAs have been set on DeathStarBench [52, 48, 93, 49] and TrainTicket [52, 36], and these papers tested various methods to scale resources for individual services. While FIRM[52] set a fine grained SLA for each service, other works explored SLAs for the system as a whole.

Interview Summary: A majority of pariticpants had an SLA defined for their organization's microservices and used it for tracking the performance of their applications. Participants did not have strict SLAs for individual services, but some used them internally for tracking performance regression.

Recommendations: While testbeds and follow-up research can represent systemwide SLAs, an ideal testbed should also include support for fine grained SLAs for each service.

3.5.5.2 Distributed Tracing

Distributed tracing is used by developers to monitor each request or transaction as it goes through different services in the application under observation. This enables them to identify bottlenecks and bugs, or track performance regression in applications in order to identify and fix the bottlenecks in them.

Academic Testbeds: All testbeds except μSuite came with a built-in distributed tracing module, whereas μSuite used eBPF for tracing the system calls made by the services. DSB-SN, DSB-HR, DSB-MR, TrainTicket and TeaStore used Jaeger as the tool used for tracing, and BookInfo used generic OpenTracing tools for the same.

Interview Summary: Only a quarter of participants used distributed tracing in their applications, and their techniques matched those used in the testbeds.

Recommendations: Given the fledgling adoption of distributed tracing in the production sphere, we recommend testbed designers leave tracing modular and easy to experiment with, and, moreover, we highly recommend this as a fruitful area for further study.

3.5.5.3 Testing Practices

Academic Testbeds: All the DeathStarBench testbeds have provisions to perform unit testing using a mock Python Thrift Client which is used for testing individual services in the application. TrainTicket also has unit testing on the individual services to check for correctness. FIRM [52] built a fault injector for DSB-SN and TrainTicket to test fault detection algorithms on these testbeds. TeaStore has a built-in end-to-end testing module for testing each service and the application as a whole. BookInfo and μSuite do not use any form of testing to test the correctness of their applications. One can use a load testing tool such as wrk2[71] to perform load test on all the testbeds except μSuite, as it uses gRPC for interfacing a frontend with a mid-tier microservice.

Interview Summary: While participants used Unit Testing to test individual components of the application, there was no consensus on the testing methods and strategies to test microservice applications as a whole. Efficient strategies for testing microservices was noted to be a pain point in various organizations, though there was an awareness of the importance of testing.

Recommendations: There is some testing framework within existing testbeds, but it has not led to clear, translatable policy recommendations for production systems. The existing testbeds cover the need for performing unit tests on individual services, but the tools for testing microservice applications as a whole is still lacking. Twitter's Diffy [94][6] allowed developers to test multiple versions of the same application in production. Researchers could use extended versions of these testbeds to implement and verify tooling around testing practices for microservices. We recommend that future testbed designers build in fault injectors, which will ideally encourage more testing-focused future work.

[6] Diffy was archived on July 1 2020.

3.5.6 Security

3.5.6.1 Security Practices

Academic Testbeds: DSB-SN, DSB-HR, and DSB-MR have encrypted communication channels by way of offering TLS support in their deployments. TeaStore and BookInfo can be deployed using Istio Service Mesh which can be configured to have encrypted communication channels between the services. TrainTicket and μSuite do not offer encrypted communication channels. None of the testbeds offer granular control or provide avenues to analyze the awareness of attack surfaces.

Interview Summary: The participants' responses showed 3 major themes regarding security in microservices: granular control, communication encryption, and attack surface awareness [74, 95]. The participants elaborated that granular control would be realized by way of having access controls implemented for a service's API to prevent attackers from gaining access to the overall system even if one service is compromised. They also cautioned about exposing too many services to the outside world, as each one would become an attack surface for entry into the application.

Recommendations: Apart from encrypting communication, the testbeds are not developed with security considerations as a design choice. There is a need for research on the appropriate security practices for microservices, both in terms of policy and the right tooling to achieve them. With the number of attack surfaces growing as the service boundaries increase, there is a need for literature on threat assessment for microservice applications.

3.6 Conclusion

Over the course of our systematization work, we arrived at a few *key insights*. The first, primary takeaway is that no extant benchmarks faithfully represent any of the production services that our participants had experience with. Although this is unsurprising, because each testbed was originally designed to investigate specific, narrowly defined questions, the lack of ready knowledge of the details of testbed limitations has given the community implicit permission to use testbeds to form conclusions about systems that are increasingly complex. While we focus on testbed mismatches, we encourage anyone investigating this area to also read the large body of microservice literature that has tackled the individual topics we address [43, 31, 73].

We also learned some surprising characteristics of current microservices from our user studies. For instance, the presence of cycles in operational, non-faulty production systems was unexpected, and indicates that the topologies the community has studied for microservices have been unnecessarily limited by outdated assumptions. Another surprising result was the overall lack of consensus between the members of our survey on simple questions such as "how would you describe microservices?". There was confusion between the role of microservices and shared libraries, indicating a need for better characterization and definitions of these terms such that the correct questions

are being asked about the correct systems. Finally, hybrid and transitional monolith-microservice architectures were shockingly common in our interview cohort, which further muddles the definitions and roles in this space.

Chapter 4

Meta's Topology and Request Workflows

4.1 Overview

Understanding microservice architectures matters for practitioners building and operating these systems, for tool developers designing debuggers and auto-scalers, and for researchers evaluating new techniques. But most knowledge about production microservices comes from high-level descriptions or small-scale testbeds, neither of which captures the true complexity of systems serving billions of users.

In this chapter, we characterize Meta's microservice architecture using 22 months of internal data. Our analysis reveals that production microservices defy common expectations. The topology is massive (12M+ instances, 180K edges) and evolves organically, with thousands of services created and deprecated monthly. Some software entities are fundamentally ill-suited to the microservice abstraction, using service identifiers in non-standard ways that obscure their true complexity. Request workflows exhibit high variability: the same parent service may call different children across executions, and concurrency patterns resist prediction from service names alone.

To understand these dynamics, we analyze both topological characteristics (Section 4.4): scale, growth, and the prevalence of ill-fitting entities, and request-workflow characteristics (Section 4.5): size, depth, predictability, and the extent to which traces capture complete executions. Our findings quantify the gap between simplified mental models and production reality, informing practitioners about the complexity they should expect and motivating more realistic benchmarks and robust observability techniques.

4.2 Toward characterizing Meta's microservices

Figure 4.1 illustrates Meta's microservice architecture. It is similar to other large-scale microservice architectures [96, 2, 7, 1, 97], consisting of ① (in Figure 4.1) a topology of interconnected, replicated software services running in dozens of datacenters; ② load balancers for distributing requests amongst service replicas; ③ an observability framework for monitoring the topology and creating traces (graphs) of a sampled set of request workflows; and ④ a globally-federated scheduler for running services on host machines within containers. A basic assumption of Meta's architecture (which may or may not be true for other organizations' architectures) is that *business use case* is a sufficient partitioning by which to define services, scale functionality, and observe behaviors.

The rest of this section motivates the value of studying the topology and request workflows, discusses limitations

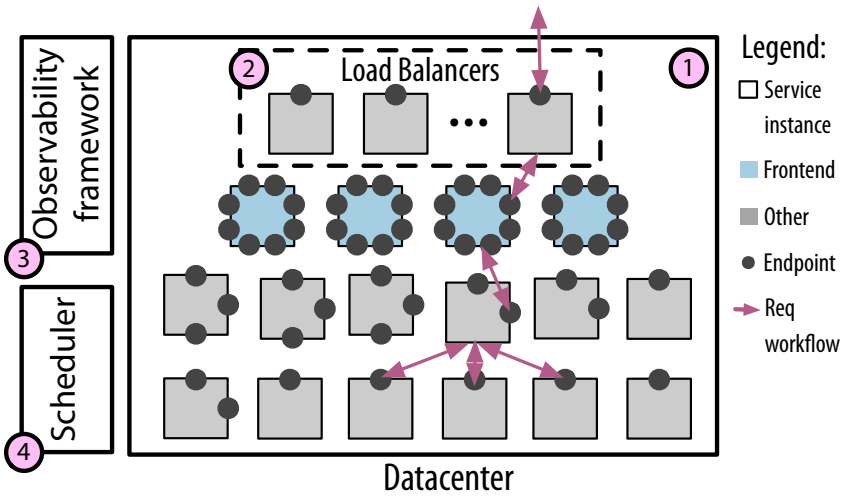


Figure 4.1: **Meta's microservice architecture and an example request workflow.** The architecture consists of many service instances distributed across many datacenters.

of previous studies, and fills in important details about Meta's architecture relevant to our analyses. We conclude by discussing the observability framework and the datasets generated from it that we use in our analyses. Given the sparsity of information available about Meta's microservice architecture, we err on the side of providing more information than strictly needed.

How do applications use Meta's microservice architecture? Customer applications, such as Instagram or Facebook mobile, issue requests that are load balanced by DNS to specific datacenters and processed by a subset of the architecture's software services. Example requests include those to save photos or record reactions to posts. Applications internal to Meta, such as dashboard or internal tools, use the architecture similarly. But, their requests are load balanced via internal mechanisms, not DNS.

4.2.1 Topology: services & communication

Why study microservice topologies? We need to understand their complexity, factors that influence their complexity, heterogeneity of constituent services, and the speed at which the topology changes. These characteristics are important to inform tools that visualize the topology, learn models based on the topology, or make assumptions about services' homogeneity [98, 99, 100, 101].

Limitations of existing studies: Only Wen et al. [96] focuses on the microservice topology. The scale they report for number of services is based on a sampled dataset of request workflows, which may not reflect the true scale of their architecture. No existing study defines what constitutes a service or how their definition impacts analyses of the topology (e.g., number of services and communication edges). Existing studies do not report on how the topology evolves or the velocity of change [2, 102].

Meta's microservice topology: The topology is formed by many replicated software services (□ in Figure 4.1) deployed across dozens of geographically-distributed datacenters along with their communication to process application requests. (Replicas are typically called *instances*). We note that within the topology, the notion of an application is ill-defined. Individual service instances may process work on behalf of multiple applications. They may also issue requests with batched data from multiple applications to other service instances. The topology evolves *organically* with no central coordination because development teams responsible for services have complete control over how they are built and maintained.

Services: Services and endpoints (● in Figure 4.1) are defined in Chapter 2. At Meta, services are named by their developers, and we use Service ID and service name interchangeably throughout this chapter.

Load-balancing & Communication: Requests to services are load balanced across their instances. Initially, a datacenter load balancer, itself a service, load balances incoming application requests. Afterward, requests between service instances are load balanced by a service-routing library [103] either linked to applications or to outbound sidecar proxies. (Only some services use sidecars, e.g., when their runtimes do not support linking the routing library directly). The routing library periodically communicates with a global service registry to discover services and routes for their instances. Requests can be load balanced to instances within the same datacenter or to instances in different datacenters. Only the datacenter load balancer is depicted in Figure 4.1.

Most services at Meta use two-way Thrift RPCs [104] for communication, with payloads serialized in Thrift binary format. Many frontend and some backend services also expose numerous HTTP (REST and GraphQL) endpoints; however, they do not have canonical names that we can use for our analyses. For this reason, we limit the endpoint analysis only to Thrift RPCs reported in the dataset from the routing library.

4.2.2 Individual request workflows

Why study request workflows: We need to understand the dynamic nature of request workflows. Given a single request execution, what will vary in subsequent executions versus what will remain stable? How much of a statement can we make about other request executions after seeing one or a limited number of samples? Such information is important to inform tools that predict performance, extract critical paths, and present aggregate analyses of request workflows.

Limitations of existing studies: Luo et al. [2] present a way to predict the total number of services that will be called at any hierarchical level of a request workflow. But, they do not discuss whether the number, set, or concurrency level of services called by a specific parent can be predicted. Wen et al. [96] present the amount of time children execute concurrently. Zhang, et al. [102] present distributions of the maximum number of concurrent

services observed in workflows. But, neither discuss if information in request workflows can predict concurrency or other workflow characteristics.

Request workflows at Meta: Requests from external applications originate at a datacenter load balancer. This load balancer sends requests to instances of *frontend services*, which are entry points for executing request business logic. There are several frontends at Meta serving different subsets of applications and each has many instances. Frontends may call many services, which in turn may call other services. The resulting hierarchy can be described as forming parent/child relationships. Request workflows for requests originating from internal applications are similar, but originate at the first service that executes business logic on behalf of them.

The set of services involved in a request workflow depends on a number of factors including (but not limited to) the business logic that must be executed on behalf of application requests and whether any requested data is cached. On the other hand, the specific set of instances involved in a request workflow depends on the current load and the load-balancing policy in use.

Sample request workflow: The arrows (↔) in Figure 4.1 show a request traversing a single datacenter. The request first arrives to an instance of the datacenter load balancer, which routes it to an instance of a frontend service, such as `www`. The request then traverses deeper into the topology to backend services.

4.3 Observability framework & datasets

Meta's observability framework includes monitoring mechanisms for recording metrics, logging mechanisms for recording various events, and a distributed-tracing infrastructure, Canopy[105], for recording graphs (called traces) of request workflows. Data generated by the framework is retained for a limited time period to reduce storage volume and due to policy. We describe Canopy in more detail below due to its criticality to observability of microservices. We conclude with a description of the log-based and trace-based datasets we use for our analyses.

Canopy for recording request workflows: Canopy works similarly to most existing distributed-tracing infrastructures [106]. It provides APIs that developers use to define a request workflow and capture important information about the workflow that should be recorded in traces. The former involves modifying services' code to propagate per-request context—e.g., request IDs and happens-before relationships—within and among the services involved in request execution. The latter involves adding *trace points*, similar to log messages, within source code. During runtime, records of trace points executed by requests are annotated with request context and timestamps. Off of the critical path of request execution, records with identical trace IDs are ordered by happens-before relationships to create traces.

Under the hood, Canopy's implementation is similar to *event-based* tracing infrastructures [107, 108, 109]. However, the way developers instrument services and use the resulting traces is similar to *span-based* tracing infrastructures [14, 110, 111]. *Implementation*: *(1)* Trace points are single events. Higher-level blocks demarcating

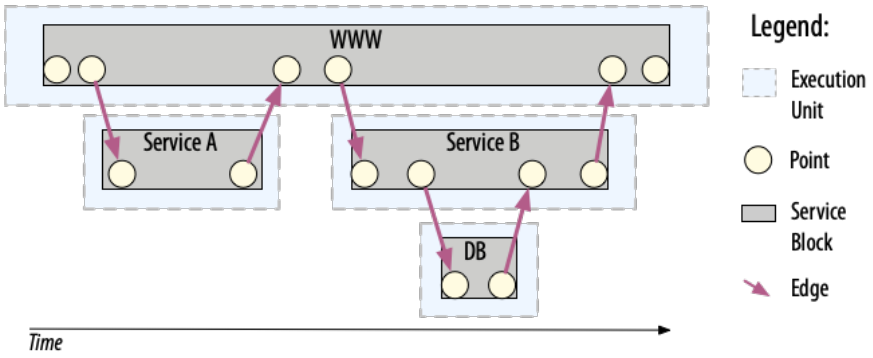


Figure 4.2: **Canopy's tracing model.**

various intervals (e.g., service executions, queuing time, or function executions) are constructed via annotations added to them. *(2)* context is propagated on both request (forward) and response (reverse) paths, allowing points to be ordered globally within and across services. *Usage*: *(1)* developers (typically) only add blocks denoting service executions; *(2)* happens-before relationships are only established in the forward direction of context propagation, meaning they identify parent/child relationships between blocks and not ordering between siblings; (3) causality between sibling blocks is not explicitly captured via alternate mechanisms. It is impossible to tell whether siblings that execute sequentially as per timestamps in one trace must execute sequentially in other traces.

We describe aspects of Canopy relevant to our workflow analyses. A key observation is that traces created with Canopy may—*by design*—not capture all of a request's workflow.

Effective trace model: Traces are graphs. Nodes are blocks (spans) indicating service execution and hierarchical levels indicate parent/child relationships. Blocks include trace points indicating message send and receives. They may contain additional points indicating other events of interest. Edges between points represent network communication. Latent work started on behalf of a request after the response is returned to the client, such as data replication or asynchronous notifications, may be recorded as additional points on the service block, or as a separate trace with a link back to the originating request's trace (similar to OpenTelemetry's *span links* [112]). Service blocks automatically record *Service ID*s and endpoint names for communication using Thrift RPCs [104]. Developers must manually provide names for services that use custom communication methods. Figure 4.2 shows an example Canopy trace originating at the *www* service. It has two children services. One of the children (Service B) also has a child (DB).

Streaming model for trace creation with timeout: A stream-processing framework [113] is used to construct traces from trace-point records for subsequent post-processing, such as computing critical path or generating end-to-end latency metrics. The framework accumulates trace events using a *session window* with a fixed gap of inactivity [114]. Traces whose events have a gap in the arrival time larger than the session window would be accumulated in more than one session, but only the first one would be used to trigger post-processing, which may result in processing of partial traces.

Dataset	Description	Format	Retention	Size	Period Used
Service History	Service deployment, lifetimes & interservice communication	*Service ID*s deployed each day	22 months	17 MB	All
Service Endpoints	Endpoints exposed by services	*Service ID* endpoints accessed each day	30 days	18.8 MB	1 day
Traces	Distributed traces	Canopy Trace Objects	30 days	13.1 PB	1 day (4.6 TB)

Table 4.1: **Datasets used for analyses.**

Per-service sampling profiles (policies) with rate limiting: Sampling profiles are unique to Canopy. They can be attached to any service and indicate sampling policies to apply based on specific attributes of incoming requests. Traces reflect the union of all sampling profiles that their corresponding requests encounter while executing. This means that a request's trace may start at a service deep in the topology, not recording prior services executed by the request. Trace branches may end prematurely at services whose profiles chose to stop recording the rest of the request's workflow.

A policy specifies: *(1)* a set of conditions for when it is applicable, such as group of endpoints, *(2)* a sampling method, *(3)* a maximum rate of trace data, measured over a sliding time window, beyond which additional traces will not be captured, and *(4)* a verbosity level to decide which instrumentation to execute for requests. Sampling methods may be random head-based sampling [110], in which requests are traced with a random probability, or adaptive sampling [115], in which the sampling probability is periodically changed to achieve a target rate of trace throughput.

Inferred service blocks: These blocks represent services that prematurely ended trace branches, either because of rate limiting or because they lacked tracing instrumentation. Inferred blocks are created during trace construction using information in parent services' message-send points. Inferred blocks may be named or unnamed. The former will be the case when parent points contain the necessary naming information.

Datasets used for this paper: Table 4.1 shows the datasets we use. For our topological analyses, we use logs describing service activity: history of deployments and deprecation, endpoints exposed by deployed services, and calls made from/to deployed services. For the workflow analyses, we use distributed traces collected by Canopy. The log data describes every deployed service, whereas traces are sampled using methods unique to Canopy, described above.

4.4 Topological Characteristics

We characterize Meta's current microservice topology as well as how it has evolved. Our analyses of the current topology uses the last (most recent) day of the *Service History* and *Endpoints* datasets (2022/12/21). Our historical analyses use all 22-months of data available to us (2021/03/01 to 2022/12/21). We also use various dashboards w/statistics about services. The main findings are summarized below.

Finding F1 (All subsections): Meta's microservice topology contains three types of software entities that

communicate within and amongst one another: *(1)* Those that represent a single, well-scoped business use case. *(2)* Those that serve many different business cases, but which are deployed as a single service (often from a single binary); *(3)* Those that are ill-suited to the microservice architecture's expectations that business use case is a sufficient partitioning on which to base scheduling, scaling, and routing decisions and to provide observability. These latter entities use *Service ID*s in custom ways, obfuscating their true complexity.

Finding F2 (§4.4.2): The topology is very complex in its current state, containing over 12 million service instances and over 180,000 communication edges between services. Individual services are mostly simple, exposing just a few endpoints, but some are very complex, exposing 1000s or more endpoints. The overall topology of connected services does not exhibit a power-law relationship typical of many large-scale networks. However, the number of endpoints services expose does show a power-law relationship.

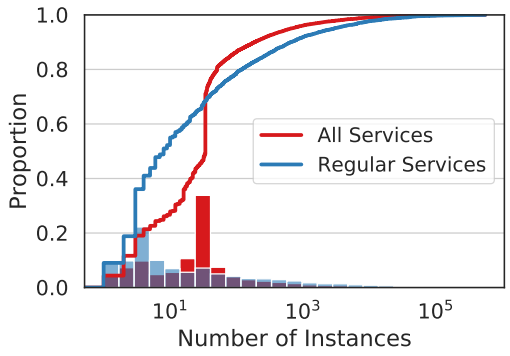
Finding F3 (§4.4.3): The topology has scaled rapidly, doubling in number of instances over the past 22 months. The rate of increase is driven by an increase in number of services (i.e., new functionality) rather than increased replication of existing ones (i.e., additional instances). The topology sees daily fluctuations due to service creations and deprecations.

4.4.1 Existence of ill-fitting software entities

We discovered several anomalous patterns in the structure of service IDs within both datasets. For example, we found that on average, 60% of services observed on any single day of the 22-month period have *Service ID*s of the form `inference_platform/model_type_{random_number}`. We found that these services all expose a small number of endpoints with identical names. Meta's engineers informed us that these *Service ID*s are generated by a general-purpose platform for hosting per-tenant machine-learning models (called the Inference Platform). The platform serves a single business use case—i.e., serving ML models—but many per-tenant use cases. Platform engineers chose to deploy each tenant's model under a separate *Service ID* so that each can be deployed and scaled independently per the tenant's requirements by the scheduler.

Following our discovery of the Inference Platform, we investigated the most frequent *Service ID*s and those with the greatest number of service instances. We found two types of software entities that use *Service ID*s in custom ways: *(1)* platforms, such as the Inference Platform, for which multi-tenancy is an additional dimension that must be considered for scheduling, scaling, routing, or observability; *(2)* storage systems, which must take into account data placement in addition to their business use case(s).

We found that some entities, such as the Inference Platform, appear as many services where each service is a combination of the business use case and the additional dimension(s) of partitioning required. Other entities, such as databases and other platforms, appear as a single service and provide their own scheduling and observability



(a) CDF

	# Instances	# Instances (Regular)
Min	1	1
Med	34	8
Avg	668	1,504
P99	9,220	31,306
Max	557,156	557,156

(b) Statistics

Figure 4.3: ***Service ID* replication factors.** The histogram is shown under the CDF. When the curves overlap, the colors are blended together.

mechanisms. Both types of entities' unique use of *Service ID*s masks their true complexity and skews service- and endpoint-based analyses of microservice topologies (ours and likely previous studies [2, **?**]).

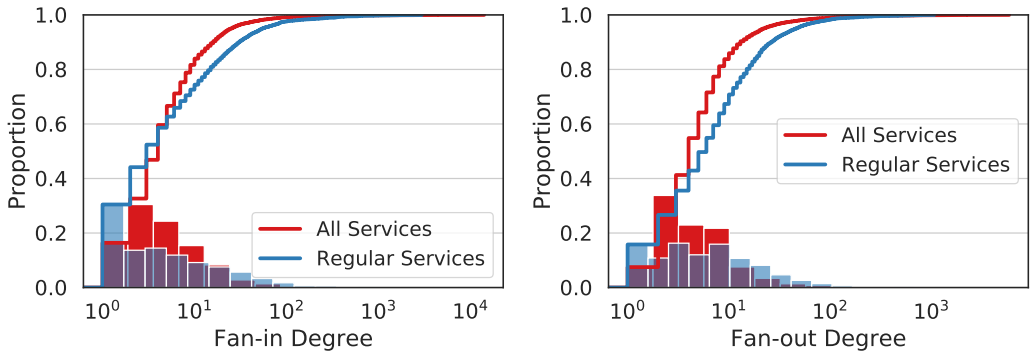
There is no systematic way to identify these ill-fitting software entities at Meta. To illustrate how they may affect *Service ID*-based analyses, we call out contributions by two entities that affect our results significantly. The first is the Inference Platform, which inflates the number of services observed. The second is the ML Scheduler, a scheduling platform for ML training jobs which chooses to appear as a single service and so inflates instance counts. We collectively refer to them and their services as *Ill-fitting services* and all other services as *Regular services*.

4.4.2 Analysis of the current topology

Scale is measured in millions of instances: On 2022/12/21, the microservice topology contained 18,500 active services and over 12 million service instances. Excluding the ill-fitting services, there are 7,400 services and 11.2 million instances.

The instance count is due to a few highly-replicated services: Figure 4.3 shows that the ill-fitting services greatly skew instance counts. Notably, the `ML scheduler` is replicated over 270,000 times, 2.2% of all instances. When these services are excluded, the median service's replication factor is only eight and the 99$^{\text{th}}$ percentile is 31,306. Frontend service `www` is the most replicated service (557,000 instances, 4.6% of all instances) as it handles most incoming requests.

Services are sparsely interconnected: We construct the service dependency diagram by connecting services that communicate with each other at least once with an edge. (Our dependency diagram is similar to that constructed by OpenTelemetry or Jaeger, except that it is constructed from a portion of the *Service History* dataset that captures communication between services, not raw traces.) There are 393,622 edges that connect services, which is much smaller than a fully connected topology (18,500^2 or 342 million edges).



(a) Fan-in CDF (b) Fan-out CDF

Statistic	Fan-in	Fan-out	Fan-in (Reg)	Fan-out (Reg)
Min	1	1	1	1
Median	4	4	3	6
Average	14	12	19	15
P99	86	101	324	158
Max	14,084	5,865	2,968	1,069

(c) Statistics

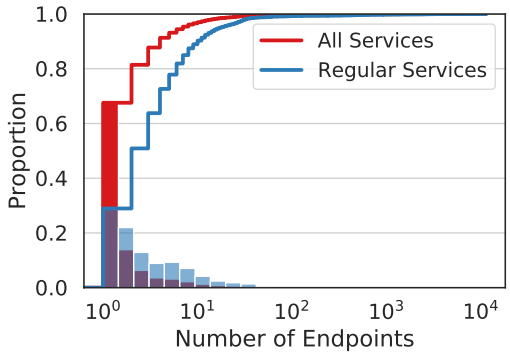
Figure 4.4: **Service fan-in and fan-out.**

Services are called by services more than they call other services: Continuing with the dependency diagram, Figure 4.4 shows CDFs and statistics about services fan-in (# of services that call them) and fan-out (# of services that they call) degrees. The median fan-in and fan-out are the same, but average and maximum fan-in is larger than fan-outs (14 vs 12 and 14,084 vs 5,865). Excluding the ill-fitting services, by removing all edges connected to ill-fitting services, decreases the median fan-in but increases the median fan-out. Excluding the ill-fitting services also increases the 99.9 percentile and decreases the maximum fan-in and fan-out values.

We investigated the services that have the highest fan-in and fan-out degrees. The former is a vault server storing credentials for use by other services. The latter is a service for querying hosts for arbitrary statistics. Both are used heavily by ill-fitting services, constituting 78% and 91% of the vault service's callers and the services called by the stats service respectively. When ill-fitting services are excluded, two other services rise to the highest fan-in and fan-out degrees respectively: a generic counting service used for various rate limiting mechanisms and a frontend service for internal applications.

Most services are simple, exposing only a few endpoints: Figure 4.5 shows a CDF and statistics of services' endpoints. Most services do not expose many endpoints (median: 1, P99: 26) and excluding ill-fitting services does not shift the statistics much. The service that exposes 11,359 endpoints is `www`, a frontend service which is used for many business use cases. It is deployed as a single binary from a large, well-engineered codebase that predates the microservice architecture.

Service complexity follows a power-law distribution: Service complexity, measured by number of unique



(a) CDF

	# Endpts	# Endpts (Regular)
Min	1	1
Med	1	2
Avg	5	14
P99	26	61
Max	11,359	11,359

(b) Statistics

Figure 4.5: **Number of endpoints exposed by services.**

endpoints in a service, follows a power law distribution ($\alpha = 2.23$, $R^2 = 0.99$), indicating that most services are simple with a long tail of more complex services. The power law does not hold for other measures of complexity. Despite there being a long tail of more complex services, the service dependency diagram does not follow a power law distribution ($R^2 = 0.62$). This means the services with more endpoints are not proportionally more connected to the topology than services with fewer endpoints. While there are some highly replicated services, the overall trend of instance counts does not follow a power law distribution either ($R^2 = 0.25$).

Sixteen different languages used to write services: Services can be written in many programming languages. There are currently 16 different programming languages in use at Meta, with the most popular being Hack (a version of PHP), measured by lines of code. Other popular languages include: C++, Python, and Java, with the rest forming a long tail.

4.4.3 Past growth & dynamism

The number of deployed service instances has almost doubled over the past 22 months: Figure 4.6 shows the percentage of deployed service instances each day as a percentage of the maximum value observed on 2022/12/21. We show different series for when all services are included, just the ill-fitting services, and only regular services. The slope when all services are considered is $s = 0.052\%$ per day (linear regression $R^2 = 0.95$). The slope when ill-fitting services are excluded is $s = 0.046\%$ per day ($R^2 = 0.95$).

The steady increase in instance counts reflects either an increase in hardware capacity over the time period or an increase in utilization of existing hardware. It cannot be explained by changes in instance sizes as they have remained mostly static.

Instances' rate of increase is due to new business use cases rather than increased scale: Figure 4.7 shows unique services deployed each day as a fraction of the maximum value observed on 2022/11/03. Note that the day with the maximum number of *Service ID*s is different from the day with the most instances. Almost all

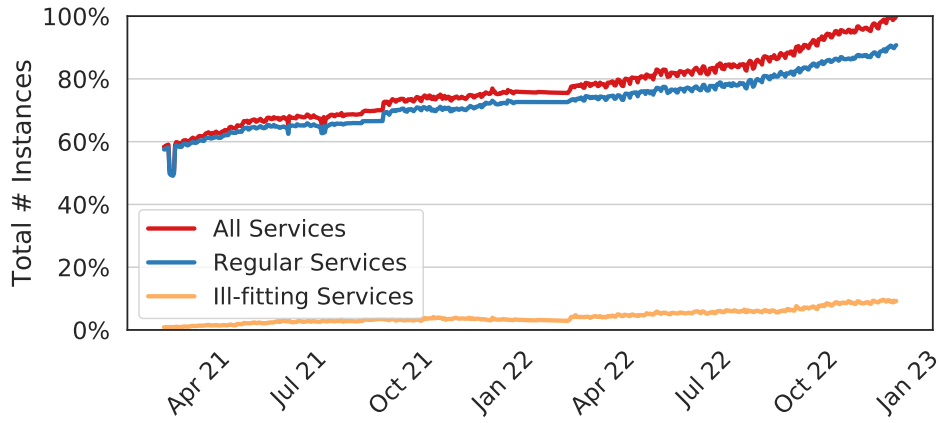


Figure 4.6: **Total service instances over time.**

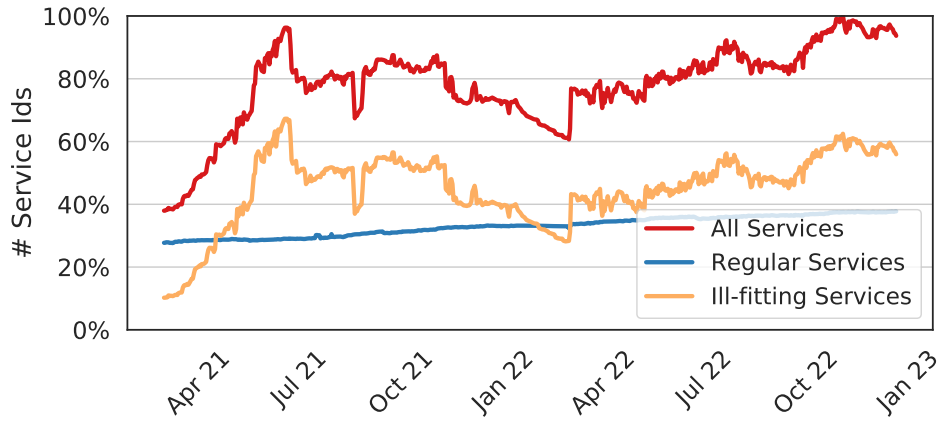


Figure 4.7: ***Service ID*s over time.**

variability is explained by the ill-fitting services, specifically the Inference Platform, which launches and terminates services as per tenants' demands. The daily increase in services when ill-fitting services are excluded (slope of *Regular Services* series) is $s = 0.043\%$ ($R^2 = 0.98$). It is almost identical to the daily increase in instance counts when ill-fitting services are excluded, which was $s = 0.046\%$ (slope of the *Regular Services* series in Figure 4.6).

Lots of churn in services, with both long-lived and ephemeral ones: Over the 22-month time period, 180,000 new *Service ID*s were deployed, 89.7% of which were deprecated at some point. Figure 4.8 shows the number of services created and deprecated each day. Newly-created services are ones whose *Service ID*s were not observed previously during the 22-month period, whereas deprecated ones are services whose *Service ID*s are never observed again. For regular services, creation rates are slightly higher than deprecation rates. As expected, ill-fitting services have high churn.

We also computed the percentage of services observed over the entire period that were deprecated in less than one week (54% of ill-fitting Services, 7.7% of regular services) and the percentage that existed throughout the 22-month period (0% of ill-fitting services, 40% of regular services).

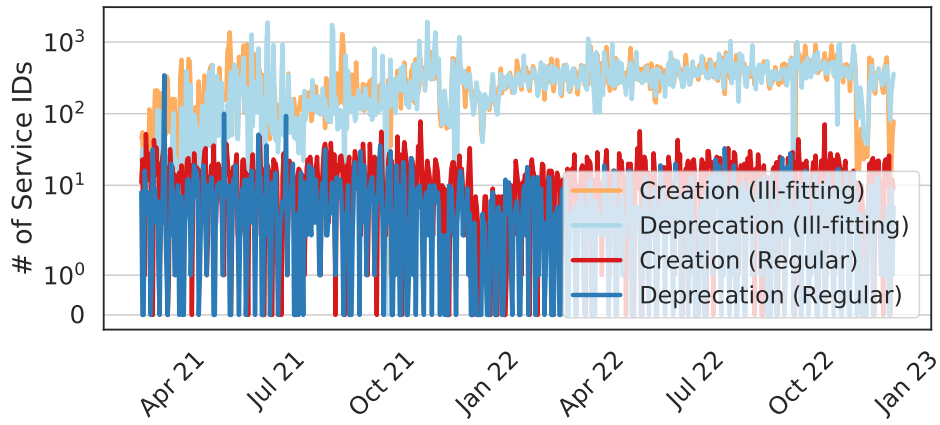


Figure 4.8: ***Service ID* creation & deprecation.**

4.5 Request-workflow characteristics

We now analyze service-level properties of individual request workflows using traces collected by different profiles. We first discuss traces' general characteristics, such as size and width (§4.5.2). We then analyze whether specific elements of a single trace predict properties of other traces representing the same high-level behavior(s) (§4.5.3-§4.5.4). As with any large-scale tracing infrastructure, traces' visibility into request workflows may be limited due to dropped records, rate limiting, and non-instrumented services. We quantify the extent to which visibility into traces is obscured as a result of these factors in §4.5.5.

Methodology: For all experiments, we use traces collected on 2022/12/21 from three profiles monitoring important high-level business functionalities. Using a few profiles and a single day avoids factors that would otherwise obscure the interpretability of our results: the effects of analyzing traces using many sampling policies and code updates that change service behaviors. Focusing on important profiles increases the likelihood that traces are representative of their workflows: the services they access are likely to propagate context accurately and use descriptive *Service ID*s and endpoint names. Overall, we analyze 6.5 million traces representing 0.5% of traces collected on 2022/12/21 by all Canopy profiles. Though we do not report them, we observed similar trends to our results on different neighboring days to 2022/12/21 while refining our experiments.

For our predictability experiments, we conduct an ex-post-facto analysis of whether *Ingress ID*s, defined as a combination of *Service ID* and ingress endpoint name, predict properties of their children across many traces. We choose to use *Ingress ID*s because they are readily available in traces, are usually the primary means of understanding trace behaviors, and are location-independent so do not require alignment of traces starting at different (unknown) depths in the topology. We do not consider global characteristics of traces, such as size or width, for prediction experiments as they are not guaranteed to be comparable due to rate limiting or dropped trace records. In our predictability sections, we mean *Ingress ID*s when we refer to parents and children, as in "unique children."

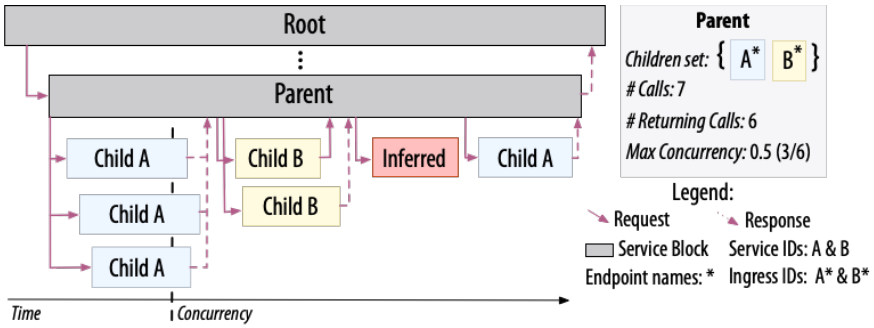


Figure 4.9: **Trace Characteristics.** Generic example trace showing attributes for a parent *Ingress ID*. Root service is either *www* or *RaaS*. *Inferred* services are represented as blocks of a fixed length since they do not contain notions of time or return edges. They are omitted from concurrency calculations.

We omit inferred calls (§4.3) from experiments that consider service names since names of inferred services are often unknown. Also, we omit *Ingress ID*s found fewer than 30 times within a profile to allow meaningful statistics to be calculated for the rest of the endpoints.

Our main findings are summarized below; we introduce the profiles afterward. Figure 4.9 describes the trace properties we analyze and predict.

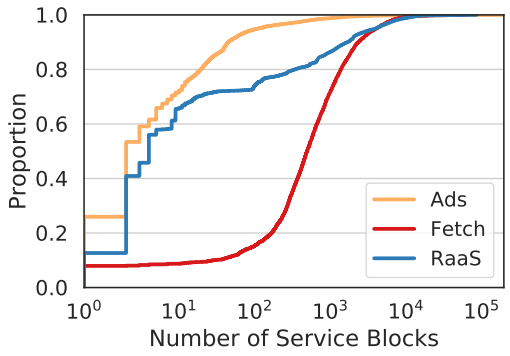
Finding F4 (§4.5.2): We measure traces with regard to the number of service blocks they contain (recall from § 4.3 that a service block represents the time interval a service was executed; repeated invocations of the same service appear as multiple service blocks.) Trace sizes vary depending on workflows' high-level behaviors, but most are small (containing only a few service blocks). Traces are generally wide (services call many other services), and shallow in depth (length of caller/callee branches).

Finding F5 (§4.5.3-§4.5.4): Root *Ingress ID*s do not predict trace properties. At the level of parent/child relationships, parents' *Ingress ID*s are predictive of the set of children *Ingress ID*s the parent will call in at least 50% of executions. But, it is not very predictive of parents' total number of RPC calls or concurrency among RPC calls. Adding children sets' *Ingress ID*s to parent *Ingress ID*s more accurately predicts concurrency of RPC calls.

Finding F6 (§4.5.5): We observe that many call paths in the traces are prematurely terminated due to rate limiting, dropped records, or non-instrumented services. Few of these call paths can be reconstructed (those known to terminate at databases) while the majority are unrecoverable. Deeper call paths are disproportionately terminated.

4.5.1 Profile details

Ads: This profile represents a traditional CRUD web application focusing on managing customers' advertisements, such as getting all advertisements belonging to a customer or updating an ad campaign parameters. The profile captures traces from 56-related endpoints exposed by the `www` frontend service. There are 3.2 million traces over



(a) CDF

Statistic	Ads	Fetch	RaaS
Min	1	1	1
Median	2	498	4
Average	83	1,052	628
P99	1,233	8,911	10,667
Max	253,877	81,859	113,790

(b) Statistics

Figure 4.10: **Trace Size.** Service block counts per trace.

the single-day period. This profile's sampling policy is random at 0.01% capped at 65 traces per second or 160 MB of trace data per minute.

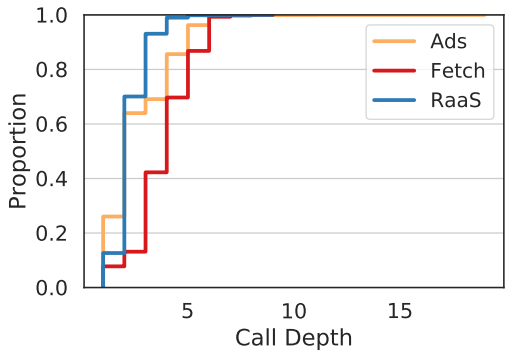
Fetch: This profile represents deferred (asynchronous) work triggered by opening the notifications tab in Meta's client applications. Examples of work include updating the total tab badge count or retrieving the set of notifications shown on the first page of the tab. It captures traces from 91-related endpoints exposed by the `www` frontend service. There are 87,000 traces over the one-day period. This profile uses adaptive sampling with a target rate of 1 trace per second, capped at 20 MB of data per minute.

RaaS (Ranking-as-a-Service): This profile represents ranking of items, such as posts in a user's feed. The RaaS sampling policy is applied to the `RaaS` service, a non-frontend service that is called by other services. As a result, traces from this profile always represent only portions of request workflows. Of the workflows we analyze, only those captured by Fetch call `RaaS`. Occasionally, a RaaS trace will be a portion of a Fetch trace, but such occurrences are rare because both Fetch and RaaS profiles use low sampling probabilities that are independent of each other. There are 3.3 million traces over the single-day period, from 4 different endpoints in `RaaS`. This profile uses adaptive sampling with a target rate of 25 traces per second.

4.5.2 General trace characteristics

There is significant diversity in trace sizes: Figure 4.10 shows CDFs and statistics of the number of service blocks in our traces. Traces collected by the Fetch profile are significantly larger than Ads and RaaS except at the tail, where Ads traces are largest.

Traces are shallow and wide: Figure 4.11 shows the maximum call depth in service blocks of our traces starting from trace roots (root is call depth 1). Figure 4.12 shows maximum trace width, which is the maximum number of calls made by all services at any depth level. (For example, 3 service blocks at one depth making 3 calls each results in a width of 9). We see that across all profiles, traces are much wider than deep: in Fetch profile the median depth is

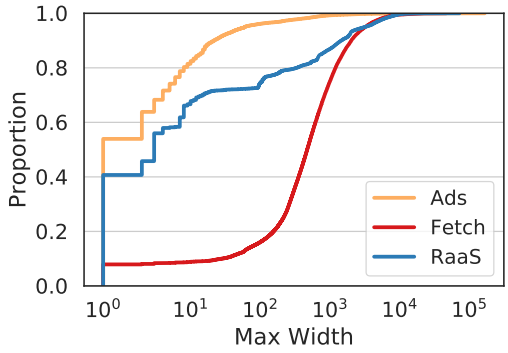


(a) CDF

Statistic	Ads	Fetch	RaaS
Min	1	1	1
Median	2	4	2
Average	2.6	3.8	2.3
P99	6	6	5
Max	19	9	9

(b) Statistics

Figure 4.11: **Call Depth.** Max depth of service blocks per trace.



(a) CDF

Statistic	Ads	Fetch	RaaS
Min	1	1	1
Median	1	472	3
Average	52	916	507
P99	774	7,395	7,885
Max	166,171	46,746	72,196

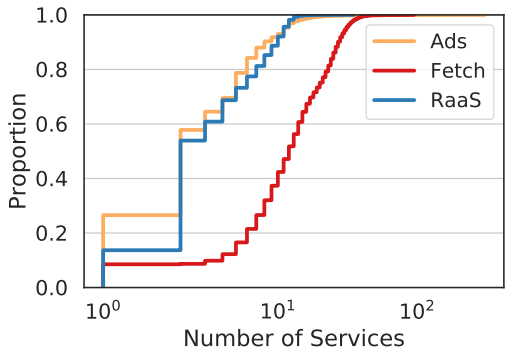
(b) Statistics

Figure 4.12: **Max Width.** Maximum number of service calls at a single call depth within a trace.

4 vs. median width of 472, and P99 depth is 19 vs. P99 width of 7,800. We conclude that large traces are a result of the number of calls made by services, not depth of calls. This can be partially explained by the widespread use of data sharding where retrieving a collection of items requires fanning out requests across many storage service instances.

Service reuse within traces is high and occurs at many different call depths: Figure 4.13 shows CDFs and statistics of the number of services visited within individual traces. Comparing with trace sizes (Figure 4.10), traces generally contain more service blocks than unique services. At the median, traces visit between 1x (2/2), 42x (503/12), and 2x (4/2) more service blocks than unique services across Ads, Fetch, and RaaS respectively; at P99 these ratios are 71x, 21x, and 810x respectively.

Most services are observed at more than one call depth in our profiles. We measured the number of call depths at which each service was observed. The services in Ads traces have high rates of appearing at multiple depths (median: 6, average: 7.3). Approximately 60% of Fetch and RaaS services are observed at multiple levels (median: 2, average: 2.6 for both profiles).



(a) CDF

Statistic	Ads	Fetch	RaaS
Min	1	1	1
Median	2	13	2
Average	4.1	14.6	4.5
P99	18	39	14
Max	307	96	36

(b) Statistics

Figure 4.13: **Unique Services.** Unique *Service ID*s in a trace.

Type	Ads	Fetch	RaaS
All	421	127	72
Leaf	168 (39.9%)	63 (49.6%)	35 (48.6%)
Single relay	58 (13.8%)	21 (16.5%)	17 (23.6%)
Variable relay	195 (46.3%)	43 (33.9%)	20 (27.8%)

Table 4.2: **Parent types.** The distribution of parents of each type within each profile.

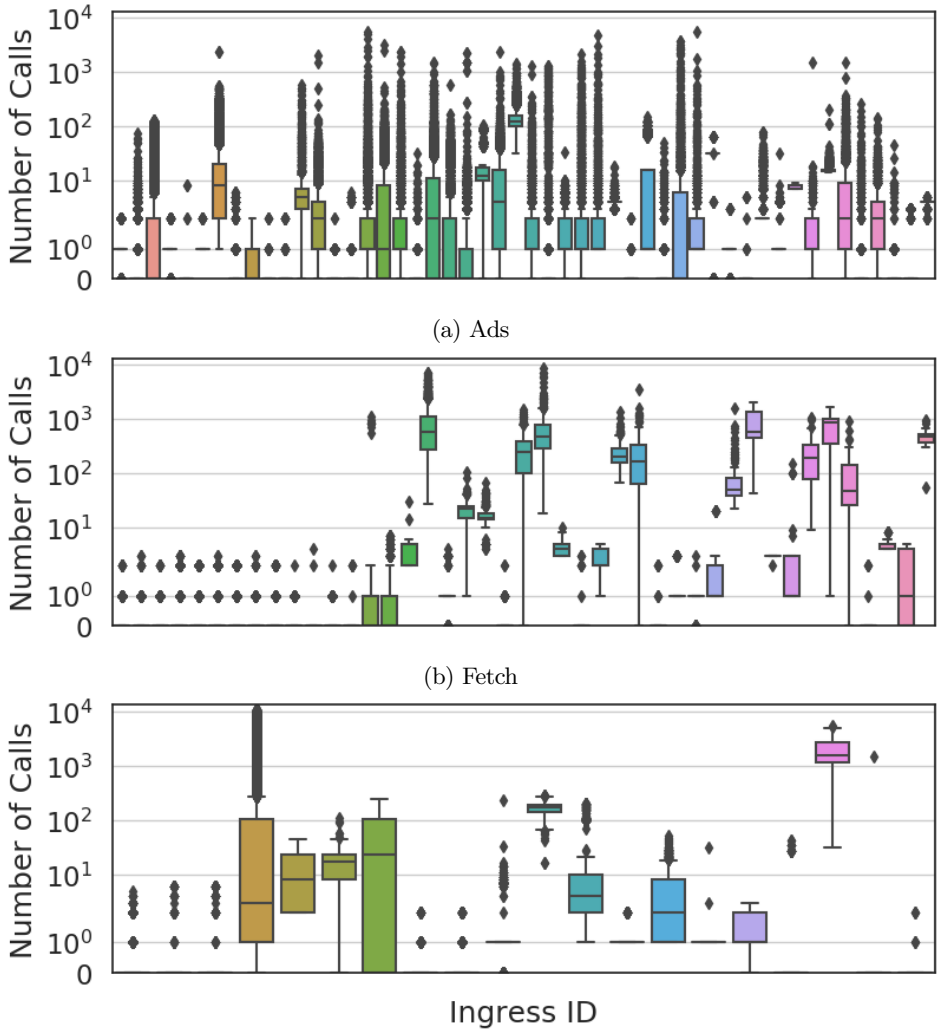
4.5.3 Predicting parent/child relationships

Parent *Ingress ID*s strongly predict whether services will have no children or only one child: Table 4.2 shows that such services, defined as *leaves* and *single relays*, make up from 53 to 73 percent of service executions in our profiles. We find that they are always databases or calls to databases.

***Ingress ID*s do not predict number of downstream calls**: Parents that make one or more downstream calls to children are called *variable relays* in Table 4.2, making up from 27 to 47 percent of *Ingress ID*s in our profiles. Figure 4.14 shows that variable relays exhibit a wide distribution in the number of children calls they make. Some *Ingress ID*s exhibit high variance in the number of children they call across different executions whereas others have very little variance (but it is always non-zero).

Variability in number of children calls is due to database calls for Fetch and RaaS: We find that at least 61.1% and 72.1% of these variable relays' children calls are database accesses in Fetch and RaaS traces respectively. For Ads traces, only 35.7% of children calls are database accesses.

There is a dominant set of unique children per parent: When we ignore number of calls, we find that most single and variable relay parents call only a few *children sets*, where each set is defined as a combination of unique children *Ingress ID*s within a given invocation of the parent *Ingress ID*. For example, one children set may contain `memcache+read` and `database+write`, whereas another may contain `key_service+retrieve` and `database+write`. The average number of children sets called by a relay parent is 28 for Fetch & Ads and 12 for RaaS parents. Most parents have a *dominant children set* that they call in more than 50% of executions.



(a) Ads

(b) Fetch

(c) RaaS

Figure 4.14: **Calls per parent.** Boxplots are shown for every Fetch and RaaS variable relay. Due to limited space, only the 50 variable relays with the greatest number of invocations are shown for Ads. Parent *Ingress IDs* are sorted in descending order by total number of invocations. Boxplot boundaries indicate P25-P75 and the horizontal line within boxes indicate medians. Lower and upper whiskers indicate the smallest/largest data values within 1.5 IQR below/above P25/P75 and dots are outliers.

Specifically, 71.9%, 80.2%, and 81.6% of Fetch, Ads, and RaaS relays have dominant children sets.

Non-dominant children sets contain mainly one off children *Ingress IDs* and are not a superset of the parent's dominant children set. On average, only 27% of children *Ingress IDs* called by a parent are in most (¿50%) of the parent's children sets.

4.5.4 Predicting children's concurrency

We define *maximum concurrency* as the maximum number of children calls executing concurrently by a parent at any point in time in its execution. More formally, a set of concurrent calls S_t at time t is all children calls with

$t_{start} \leq t < t_{end}$, where all timestamps are measured at the parent, and the maximum concurrency C is computed as:

$$C = max(|S_t|), \forall t : t_{start}^{parent} \leq t < t_{end}^{parent} \tag{4.1}$$

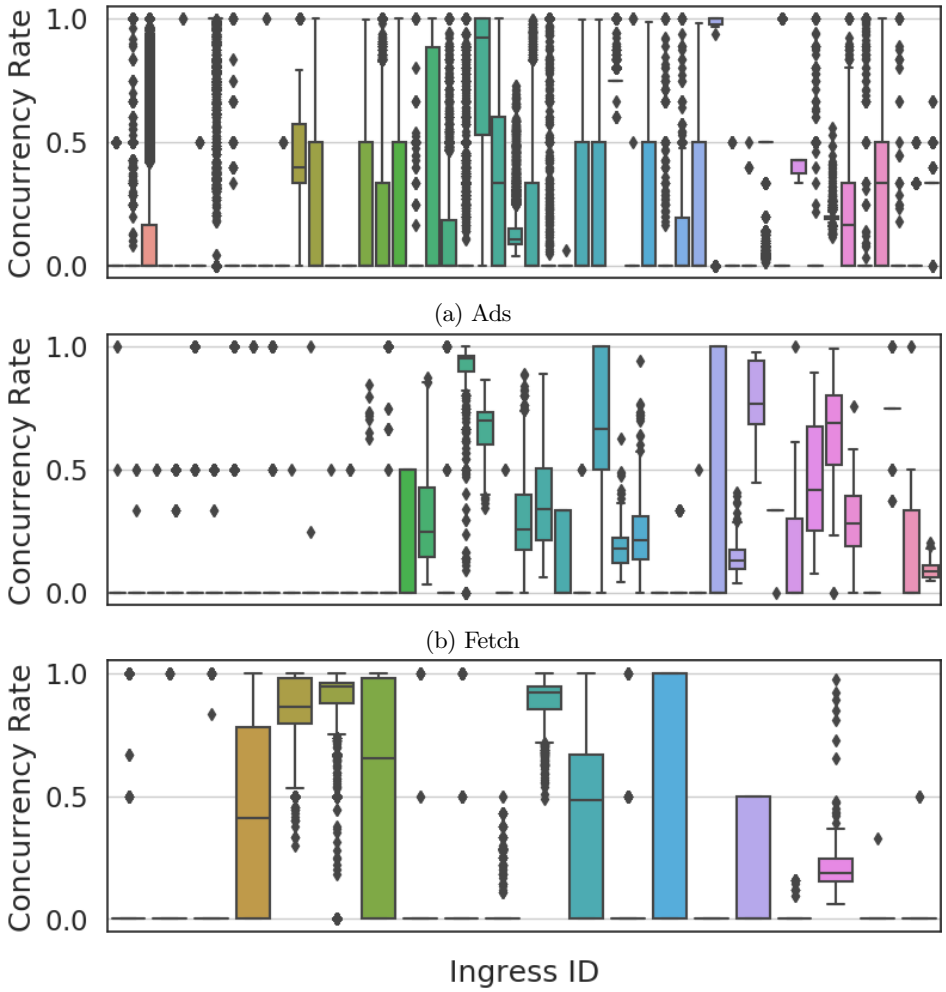
We use a normalized measure of maximum concurrency, the *concurrency rate*, calculated as $C/num\_children$, to allow comparisons across different executions of the same parent (different numbers of children may be called in each execution) and to allow comparisons between different parents. *num_children* refers to children that have return edges and well-defined durations. We only consider variable relays since concurrency is ill-defined for leaves and single relays.

Parent *Ingress ID* does not predict whether children will execute concurrently or sequentially: Figure 4.15 shows boxplots of concurrency rates across executions for each parent *Ingress ID* observed in our traces. We see that there is a mix of high and low variation in concurrency rate across *Ingress ID*s.

The combination of parent *Ingress ID* and children set more accurately predicts concurrency rate: Figure 4.16 shows a CDF of the standard deviation in concurrency rate across all executions of parent *Ingress ID*s. To understand if children set adds predictability value, we calculate the standard deviation for each parent's children set and average them to obtain a per-parent average. We plot this CDF of per-parent average. Intuitively, if children sets provide value, the per-parent average should decrease whereas if they do not, the data points will be randomly distributed and standard deviation will not decrease. Overall, we find that including children sets shifts the distributions to the left. The shift is most pronounced at the median for Ads and Fetch: 0.13, 0.09 vs. 0.04 and 0.02. Adding children set does not provide value in the tail for Fetch and RaaS.

We speculate the reduction in standard deviation is because children belonging to the same children set likely have well-defined control or data dependencies between each other. Reduction in variation due to control dependencies may be a result of custom threading models for different code logic blocks in parents (each responsible for a different behavior and thus children set). For data dependencies, consider the following examples. Children sets containing different cache services may have no dependencies and thus may be able to execute concurrently. In contrast, children sets comprised of a key server and a database service may have to execute sequentially: credentials may be required to access the database.

***Ingress ID* + children set calls display a range of dependency relationships**: We now quantify the strength of dependencies within *Ingress ID* + children set's calls. We use the maximum concurrency rate observed across *Ingress ID* + children set executions as a indicator of dependency. A maximum concurrency rate of 1 implies that there are no dependencies among children calls. A maximum observed concurrency rate of 0 builds confidence that the children calls are dependent and must execute sequentially. Figure 4.17 shows the results. Overall, we find that most *Ingress ID* + children set executions display weak dependencies (some concurrency) and there are



(a) Ads

(b) Fetch

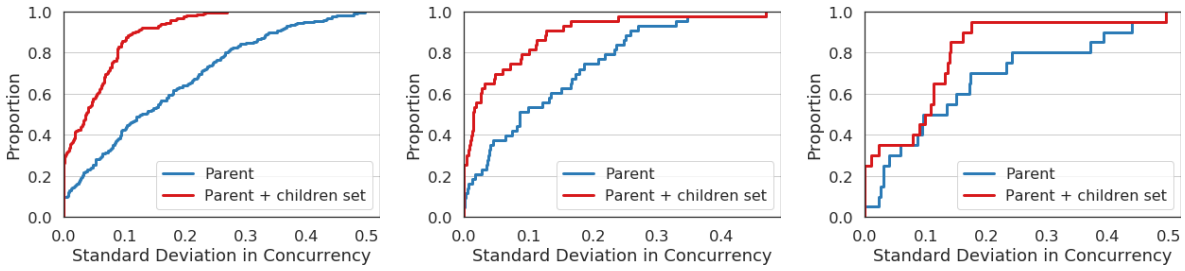
(c) RaaS

Figure 4.15: **Parent concurrency.** Concurrency distribution for all invocations of a parent *Ingress ID*. Shows all variable relays for Fetch & RaaS, and the top 50 in Ads by invocation count. Boxplots are interpreted identically to figure 4.14.

a few strongly dependent (sequential) children sets.

4.5.5 Quantifying traces' observability loss

Most prematurely terminated call paths are unrecoverable: We plot the percent of branches that terminate prematurely (at an *inferred* service) at each call depth in our traces (Figure 4.18). Some branches terminate prematurely due to internal rate-limiting at databases, which are usually leaves in the traces. The shaded area in Figure 4.18 is the portion of inferred services that represent known databases. The distance between the curves are unknown inferred services, which make up the majority of inferred services. Using trace data alone, we cannot know what the unknown inferred services are (some may still be other databases we were not able to identify reliably) or the shape of the workflow from that point on.



(a) Ads (b) Fetch (c) RaaS

Figure 4.16: **Standard deviation in concurrency rate.** *Parent* shows a CDF the standard deviation in concurrency rates for all executions of a parent. *Per-parent avg. children set* shows the average standard deviation per children set for each parent.

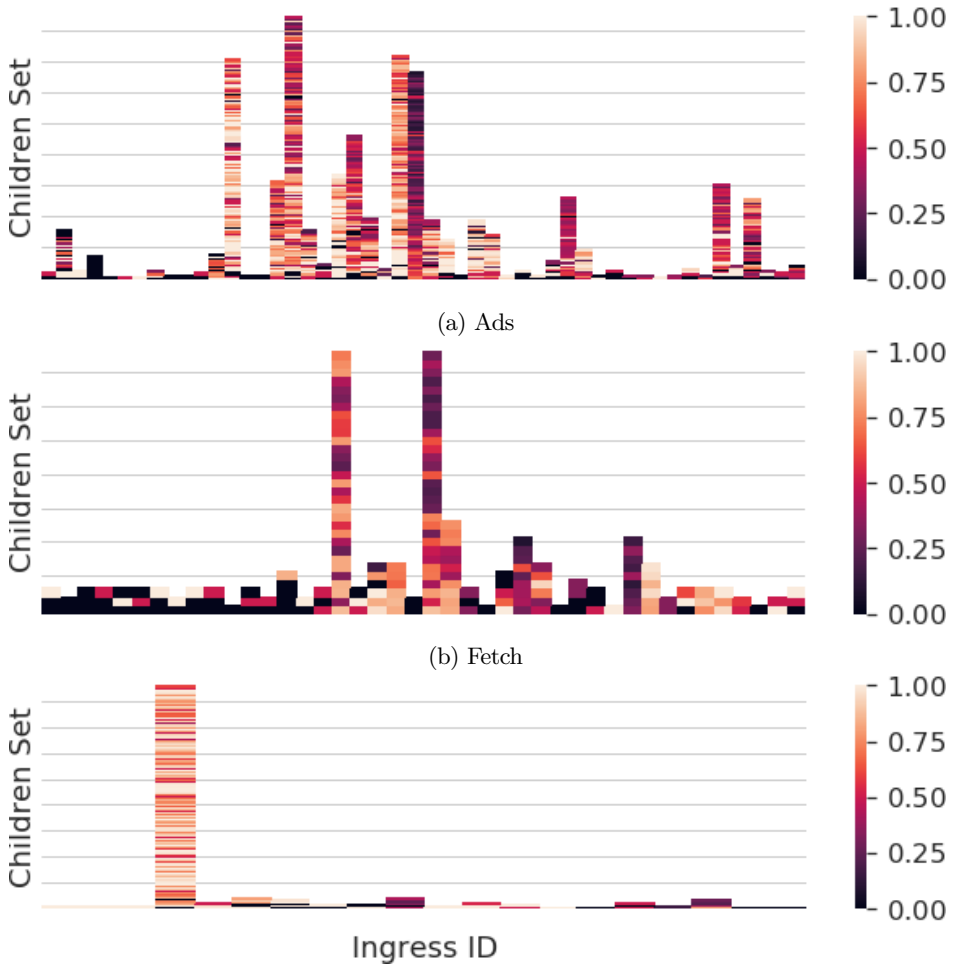
Non-uniform probability a branch is terminated: Deep branches are disproportionately prematurely terminated. For RaaS traces, 80% of call paths that reach depth 3 are terminated with inferred nodes, none of which were identified as databases. However, as the average trace depth for RaaS is only 2.3 (Figure 4.11), the majority of RaaS traces are not affected by premature branch terminations. Similarly, Fetch and Ads traces are shallow and prematurely terminated branches mainly occur beyond the average trace depth.

4.6 Implications and opportunities

Implications for microservice testbeds: Existing testbeds [4, 5, 45] represent only single applications, whereas microservices within Meta serve many applications (§4.2.1). Previous studies state that existing open-source testbeds' topologies are lacking in scale compared to industrial microservices [2, 96]. Our results confirm these results (Finding F2) and add the following dimensions to consider in future testbeds: heterogeneity of services, churn, and growth. Specifically, we find that Meta's microservice architecture contains a mix of software entities that are deployed as services: complex ones that expose many endpoints and are likely more monolithic in nature, simple ones that expose just a few, and ill-fitting ones that require support beyond which the microservice architecture provides by default (Finding F1). We find that services are deployed and deprecated (at least) daily and that the shape of the communication topology is constantly growing and changing (Finding F3).

Luo et al. [2] state that request workflows within existing testbeds are too static. Many service-level workflow properties can be predicted from root endpoint alone. Our analyses show that future testbeds should include concurrency, number of children, and set of children that are executed as dimensions of variability in request workflows representing the same or similar high-level behaviors (Finding F5).

Implications for microservice tooling: Tools that use models of microservice topologies [98, 99, 100] should assume that its constituent services are always changing and that the topology itself is highly-dynamic (Finding F3).



(a) Ads

(b) Fetch

(c) RaaS

Figure 4.17: **Parent *Ingress ID* + children set max concurrency rate.**

Periodic retraining may be necessary; mechanisms are needed to identify when predictions diverge from the ground truth due to stale topological information.

Tools that aggregate request-workflow traces for performance predictions, diagnosis, or capacity planning [108, 101, 116, 117, 102] must assume that there is significant diversity in workflows originating from the same root endpoints or groups of related root endpoints (Finding F5). Our studies show that many workflow properties can be predicted when they are broken down into fundamental building blocks (parent/child relationships) (Finding F5), perhaps a promising starting point for aggregation-based tools. However, capturing total orderings for entire traces [101] or even individual services may not scale due to parent *Ingress ID*s initiating large number of RPCs with high concurrency (§4.5.4).

Need for artificial microservice topology & workflow generators: Such generators are a necessity given the infeasibility of creating microservice deployments outside of industrial settings. The sole existing workflow generator [2] may be too specific to a single organizations' microservice design (that number of children depends on

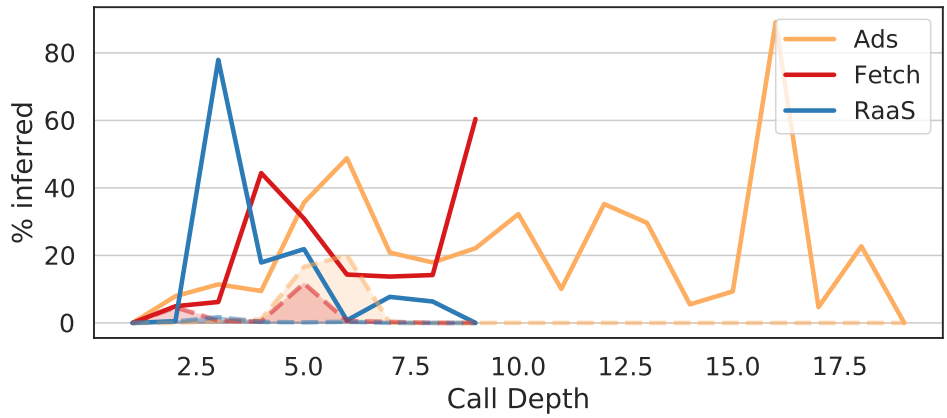


Figure 4.18: **Inferred Services.** Percent of service calls that are inferred at each call depth. The shaded region is the percent of inferred services that are known to be database calls.

depth in trace) and generates stochastic workflows that do not represent any single request. Research is needed to identify: *(1)* which dimensions of microservice architectures are best explored in testbeds versus artificial topology or workflow generators; *(2)* how to ensure these dimensions are representative of a variety of large-scale organizations' characteristics. Our analysis shows that assuming topologies follow power-law relationships is insufficient for modeling microservice topologies (Finding F2).

How to better incorporate ill-fitting software entities into microservice architecture? Ill-fitting entities constitute a significant portion of Meta's microservice topology. Key questions include: *(1)* Should infrastructure platforms provide richer interfaces to allow scheduling, scaling, and observability based on additional dimensions rather than only one? *(2)* In cases where ill-fitting-entities use custom techniques, what mechanisms are necessary to allow mapping them to standard service-level operations?

Naming & predicting missing elements of workflows: Our predictability results (Finding F5) indicate that well-defined service and endpoint names are important for predicting local workflow properties. Almost all tools that use distributed traces [108, 101, 116, 117, 102, 98, 99, 100, 118] assume descriptive names. But, naming quality can vary considerably, especially for services that satisfy many business use cases and for microservice architectures in which all instrumentation is done within proxies surrounding services [46]. Research into naming schemas that allow different parts of service behaviors to be differentiated based on parts of the name (or attached attributes) is needed. Research is also needed into how to automatically identify meaningful names and/or attributes that differentiate important within-service behaviors, and whether missing observability data (Finding F6) can be predicted based on other data already available.

Need for standardized methods to contrast different organizations' microservice architectures: Our original goal for this research was to compare characteristics of Meta's microservice architecture with previous studies of industrial microservice architectures. At 30,000 ft, we find that organizations' architectures have similar

architectural diagrams (Figure **??**) and use custom versions of the same architectural components or open-source versions [119, 14, 104]. Furthermore, similar to Meta, the traces used in Luo et al. [2] and Wen et al. [96] tend to be small. Large traces are wider than deep, indicating common use of data sharding. We also find some differences. Traces used in Zhang et al. [102] seem to be much deeper than those used in our analyses, perhaps due to their domain-oriented microservice strategy [120].

Unfortunately, we found more detailed quantitative comparisons to be impossible due to divergent (or ill-specified) definitions in previous studies and because different studies use custom measurement techniques specific to their observability frameworks. With regard to comparing scale and complexity, previous studies do not define the term *service*, describe individual service's complexity, or describe number of communication edges between services, or service instances. For request-workflow-based analyses, these studies do not identify tracing sampling rates and mechanisms, whether traces capture all of request workflows or only parts or whether dropped records or rate limiting impact their analyses. Similar to rich research into Internet measurement [121], we need to develop rich, well-accepted methodologies for collecting data about microservice architectures to understand and systematize similarities and differences across them.

Chapter 5

Alibaba's Trace Data Quality

5.1 Overview

Distributed tracing is the primary tool for understanding how requests flow through microservice architectures. Practitioners use traces to debug performance issues, identify bottlenecks, and build automated management tools. Researchers use public trace datasets to develop and evaluate new techniques. But what happens when the trace data itself is unreliable?

In this chapter, we examine Alibaba's publicly-released trace datasets, 12 hours of data from 2021 and 13 days from 2022[122, 123], which have been widely used by the research community [8, 124, 9, 125]. Our analysis reveals pervasive data quality issues: 99.48% of traces in the 2021 dataset and 85.77% in the 2022 dataset violate at least one of Alibaba's stated specifications. Inconsistencies include missing rows, contradictory field values, and context-propagation errors where services fail to correctly increment call identifiers. Naive trace reconstruction following the stated specifications produces smaller, disconnected graphs that misrepresent the original request workflows.

To address these issues, we develop CASPER, a reconstruction tool that exploits hidden redundancies in the data model to recover from both data loss and context-propagation errors. Section 5.2 describes the dataset format and construction process. Section 5.3 catalogs the inconsistencies we discovered and their root causes. Section 5.4 presents CASPER's reconstruction algorithms, and Section 5.5 evaluates their effectiveness, recovering traces that are 1.14–2.71× larger, 1.22–1.32× deeper, and increasing the fraction of fully-connected traces from 58% to 84%.

These findings have implications beyond this specific dataset: they demonstrate that trace data quality cannot be assumed, and that tools built on traces must account for incomplete and inconsistent inputs.

5.2 Alibaba microservice datasets

This section describes the Alibaba datasets' tabular format and the specifications describing traces that are stored within in them (§5.2.1). We also describe how traces can be constructed from the tabular data using the specifications (§5.2.1). §**??** discusses how the datasets are inconsistent from the specifications and their impact on traces constructed assuming consistency.

We start with a brief description of Alibaba's distributed-tracing infrastructure, which was responsible for

capturing the traces. Please see Luo et al. [2] and the datasets READMEs [122, 123] for a more detailed description of the tracing infrastructure and the datasets respectively.

Alibaba's distributed-tracing infrastructure for capturing request-workflow traces: Like most distributed-tracing infrastructures [14, 105, 110], Alibaba's infrastructure works by propagating context with requests' execution. For Alibaba, context includes a per-request unique ID (`traceid`) and a per-call path unique ID (called the `rpcid`). The `traceid` uniquely identifies calls made on behalf of a single request. The `rpcid` uniquely identifies calls and their depth within the trace call graph. It is specified as a series of delimiter `'.'` + IDs. Each service adds a delimiter and adds a unique ID before calling a downstream service. The number of delimiters is equal to the depth of the call. When requests execute log messages, records of them are enriched with context and stored in long-term storage. The trace datasets are comprised of the subset of these logs indicating caller/callee relationships.

5.2.1 Tabular format & storage specifications

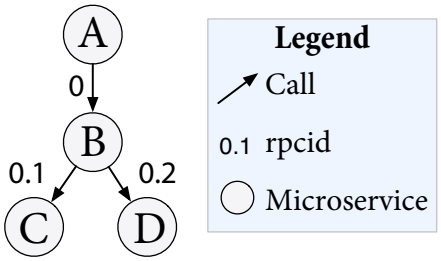
Format: Figure 5.1a shows a simplified version of the tabular format, which is largely the same for both datasets. It also shows the graph representation of the trace in Figure 5.1b. Various columns provide information needed to create the trace topology (nodes and edges) and add annotations. Rows represent log messages or edges of the trace graph—i.e., communication calls between an upstream service (caller) and downstream service (callee). Up to two rows may correspond to a single communication call.

Specifications: The `traceid`, `rpcid`, `um`, and `dm` columns encode traces' topological information. The first three fields are propagated in context as described above. For a given call the `um` and `dm` fields identify the corresponding upstream service (caller) and downstream service (callee).

The remaining columns are used to annotate trace nodes or edges. We discuss only ones relevant to our analyses. The `rpctype` column describes the protocol used for a given call. It can be either *RPC*, *HTTP*, *mc* (Memcache), *mq* (Message queue), or *db* (database). The `rt` column describes the call's response time and `ts` denotes a timestamp indicating when the row was recorded by a service. There are two rows for each RPC and HTTP-based call. The first

	Topology				Annotations	
`ts`	`traceid`	`rpcid`	`um`	`dm`	`rpctype`	`rt`
166370	1	0	A	B	http	8
166374	1	0	A	B	http	-7
166376	1	0.1	B	C	db	0
166372	1	0.2	B	D	mc	0

(a) An encoded trace in tabular form



(b) Corresponding constructed trace

Figure 5.1: **A trace in tabular form & its constructed version.** Not all annotations are shown in the tabular version. Annotations are omitted from the graph. Table follows the 2021 data format.

row records the end-to-end response time as measured by the upstream service. The second row records the processing time of the request within the downstream service—i.e., the latency between receiving the request and sending a reply.

Differences between 2021 and 2022 datasets In the 2021 dataset, the row corresponding to end-to-end latency records a positive response time whereas the row corresponding to downstream processing records a negative one [122]. Both response-time values are positive in the 2022 dataset [123]. The 2022 dataset includes additional annotation fields.

5.2.2 Constructing traces

The construction process described below is general to accommodate traces that start at any arbitrary point of request workflows' execution (i.e., not necessarily at frontend services where workflows typically originate). Responses to questions about the datasets from Alibaba indicate such intermediate starting points are possible [126]. Figure 5.1b shows the trace that would be constructed from the tabular data in Table 5.1a.

To build a trace: *1)* Extract all rows of the table with the same `traceid`. *2)* Group rows with the same `rpcid` together as they represent the same call. *2)* Find roots, which are named by the `um` or `dm` field of calls with the fewest number of `'.'` delimiters in their `rpcid` values. The `UM` is the root if it is defined, else `dm` is the root. The former accommodates intermediate starting points and the latter frontends. *3)* Find calls made by roots, which have the same `rpcid` prefix as roots with one extra `'.'` delimiter, and attach their `dm` values as children. *4)* Repeat step 3 recursively for all leaves in the trace until there are no remaining calls left. Nodes and edges can be optionally annotated during this process.

5.3 Trace Inconsistencies

We identified four types of inconsistencies in the Alibaba trace datasets that invalidate the assumptions mentioned in §??. We found these inconsistencies to be prevalent within the datasets with almost all traces having at least one inconsistency. For the remainder of this section, we discuss inconsistencies within the context of a single trace. We focus mainly on the 2021 trace dataset since it has higher error rates, but all inconsistencies discussed were also observed in the 2022 dataset.

Analysis of Alibaba's responses to questions about the datasets [127, 128, 129] indicate that these inconsistencies are explained by data loss and context-propagation errors (CPEs). Data loss results in rows being dropped or fields in rows missing values. Context propagation errors occur when a user does not correctly increment the `rpcid`, assigning the same `rpcid` to many calls. This non-unique `rpcid` is propagated downstream, resulting in downstream calls also having non-unique `rpcid`s. The last subsection (§5.3.5) gives an example on how many of these inconsistencies may arise given a context propagation error.

Inconsistency	2021	2022
Missing duplicate row	99.48%	85.77%
Missing `rpcid`	35.23%	11.42%
Unexpected row	30.16%	6.86%
Contradicting `um`	33%	7.94%
Contradicting `dm`	26.84%	2.56%
Missing value	94.2%	67.05%

Table 5.1: **Inconsistency frequencies.** The portion of traces that have at least once occurrence of each inconsistency.

Table 5.1 lists the percentage of unique traces affected by each inconsistency. We next describe each inconsistency in detail.

5.3.1 Missing rows

There are many missing rows in the trace datasets. We categorize missing rows into two groups: missing duplicate rows and missing `rpcid`s. The duplicate row (i.e. the call or reply) for two-way communication (*http* or *rpc*) is often missing, leaving only one row with either a positive or negative `rt` value. Most traces are missing a duplicate row in both datasets.

A missing `rpcid` is defined to be when we are missing all rows for a `rpcid`. Since `rpcid`s encode topological information about the request workflow, we know all rows for a `rpcid` are missing when we are missing a `rpcid` that is ancestrally between two captured `rpcid`s and is needed to form a call path. We call these missing `rpcid`s internal `rpcid`s (since they are internal nodes in a call graph). A trace may have missing rows before the smallest `rpcid` or after the largest `rpcid`, but there is no way to detect this.

Example: Table 5.2 gives an example of a missing row and `rpcid`. We are missing the negative `rt` row for the *http* request 0.2. Additionally, we are missing all rows for the `rpcid` 0.2.1, which is the connection between 0.2 and 0.2.1.1.

Implication: Missing duplicate rows does not impact the shape of the trace since the remaining row contains the call path information. We only lose the `rt` for one direction of communication. When we are missing `rpcid`s in a

`rpcid`	`um`	`dm`	`rpctype`	`rt`
0.2	A	B	http	+
0.2.1.1	C	D	db	+

Table 5.2: **Missing rpcids.** The rpcid 0.2.1 is missing from the table since it's needed to connect 0.2 and 0.2.1.1. Additionally, a duplicate row is missing for the *http* request 0.2.

rpcid	um	dm	rpctype	rt
0.3	A	B	http	+/-
0.3.1	B	C	rpc	+/-
0.3.1	B	C	rpc	+/-
0.3.1	B	D	mq	+
0.3.1.1	C	E	mq	+

Table 5.3: **Unexpected rows.** `rpcid` 0.3.1 is repeated above the expected threshold for the `rpctype`. +/- *rt* is used to indicate we have both the positive and negative rows.

trace, naive rebuilding (using the assumptions outlined in §**??**) would result in a disconnected trace, under-counting the number of calls and not preserving the true topology.

How to address the inconsistency: Data loss causes us to lose rows, which can present as either a missing duplicate row or a missing `rpcid`. We can use redundant information about the structure of a trace in the `rpcid`s to replace missing internal `rpcid`s, when it's available. For example, in Table 5.2, the missing `rpcid` 0.2.1 should have `um` B and `dm` C to form a valid call path.

5.3.2 Additional unexpected rows

As described in Section **??**, `rpcid`s are assumed to be unique for each call in the system. We expect to see one row per message sent; for two-way communication, there should be two rows (call & reply) and for one-way communication there should be one row. Additionally, when we have a reply row for an `rpcid`, all structural information (e.g. `um`, `dm`, `rpctype`) should be identical since it references a single call. Despite this assumption, we often see additional rows past these thresholds for a single `rpcid`. In the 2021 traces, 42.18% of traces have at least one `rpcid` with unexpected rows.

Example: Table 5.3 shows an example of additional unexpected rows where 0.3.1 is repeated many times. We have four rows to `dm` C (counting both the + and - `rt` rows) and one row to `dm` D.

Implication: The assumption that `rpcid`s are unique is invalidated and rebuilding the trace naïvely would under-count the number of unique calls. Additionally, when there are multiple `um`, `dm` pairs, it's not clear which should be used for the `rpcid`.

How to address the inconsistency Additional unexpected rows are caused by context propagation errors (CPEs), where the user of the tracing infrastructure did not increment the `rpcid` for each unique call. This appears in the table as additional rows with the same `rpcid` and `um`, but potentially different `dm`s. In Table 5.3, the CPE originates from service B, which makes at least two calls to service C and one to D. The non-unique `rpcid`s are passed downstream, creating more non-unique `rpcid`s (and call paths) further downstream. For example, 0.3.1.1 (in Table 5.3) has `um` C, but we do not know which of B's calls to C made the subsequent call to E.

rpcid	um	dm	rpctype	rt
0.3.1.1	C	E	mq	+
0.3.1.1	D	F	mq	+

Table 5.4: **Contradicting values.**

5.3.3 Contradicting Values

There are inconsistencies in the datasets where rows that should contain identical values have conflicting values (e.g. the two rows corresponding to the call and reply for a two-way communication call should have the same `um` and `dm`). We categorize contradicting values into two groups: contradicting `dm`s are when all rows with a `um` has multiple `dm`s and contradicting `um`s are when one or more of the `um`s for an `rpcid` don't match the upstream call's `dm` (i.e. not forming a valid call path). Contradicting `um`s are independent of the `dm` value. A single `rpcid` can have both types of contradicting values in their rows. A large portion of the 2021 traces (26%) have at least one `rpcid` with contradicting `dm` values and 37% of the traces have contradicting `um`s.

Example: Table 5.4 shows two rows for `rpcid` 0.3.1.1. There is conflicting `um` and `dm` information in the two rows. Since there are multiple `um` values, at least one of the rows may not connect upstream resulting in an invalid path.

Implication: Naïvely rebuilding trace with contradicting values could create invalid call paths, depending on which row's information is added to the trace topology. Since each unique `rpcid` is assumed to have one `um` and one `dm`, naïvely rebuilding would not check for this inconsistency.

How to address the inconsistency Contradicting values are the result of context propagation errors. Contradicting `dm`s appear when the user does not increment the `rpcid` when making calls to different downstream services. Upon cursory inspection, we found contradicting `um`s are either downstream from CPEs or seem to have an incorrect `rpcid` that could be remedied by adding a '.' followed by an integer to connect the call one level downstream. We can use redundant information about the call paths to help determine accurate `um`s and `dm`s (or if the `rpcid` is not unique).

5.3.4 Missing Values

Many values in the datasets are missing or contain `'(?)'/UNKNOWN` as the value. In fact, most traces in both datasets contain at least one missing `um` or `dm` value (94.2% and 69%).

Implication: Naïvely rebuilding traces with missing values misses opportunities to uncover the true value, resulting in skewed metrics about the frequency of specific microservices.

How to address the inconsistency Missing values are the result of data loss, which is not uncommon in large distributed systems. For two-way communication, there is often a duplicate row for the `rpcid` which contains the missing information. If there is no duplicate row, these missing values can be recovered using call path information from an upstream or downstream call.

5.3.5 Combination of inconsistencies

Context propagation errors (CPE) often show up as a combination of the inconsistencies. We always see unexpected rows for CPEs, but this is typically combined with contradicting values. For example Table 5.3 showed both `dm` values C and D, which are contradicting.

Downstream from CPEs, the same `rpcid` is used as a seed for downstream calls. In the Table 5.3 example, 0.3.1 is the seed `rpcid` for all downstream calls made from C and D. If we added an additional row to this example with `rpcid` 0.3.1.1.1, `um` X and `dm` Y, we would have a contradicting path (since X is not the same as upstream call 0.3.1.1's `dm` E). To make things more complicated, the contradicting path inconsistency would no longer exist if we assumed we were missing its' upstream `rpcid` (which could have `dm` X). The point here is that CPEs cause many inconsistencies since they invalidate the assumption that `rpcid`s are unique. This makes it challenging (and sometimes impossible) to decipher the trace topology.

5.4 casper

We introduce CASPER, which aims to create the largest accurate request workflow topologies. Building on the intuition in §**??**, we discuss which inconsistencies can be circumvented (e.g., are recoverable), and how to recover from them (§5.4.1). We also discuss when inconsistencies are not recoverable (§5.4.2). We then present CASPER's algorithm for rebuilding the traces (§5.4.3). We conclude with limitations of our approach (§5.4.4). CASPER is implemented in 711 lines of Python code. It takes as input all rows for a trace and outputs the constructed trace in Alibaba tabular or OpenTelemetry JSON [14] format.

5.4.1 Recoverable inconsistencies

5.4.1.1 Data loss

Missing internal calls: Observing missing internal calls due to data loss, we can determine the exact number of missing calls on the call path using the `rpcid` structure. We find the call that is missing its upstream call and recursively drop the trailing '.' from the `rpcid` until we reach an existing `rpcid`. Each new `rpcid` we create by dropping a '.' becomes a call in the trace.

Example: In figure 5.2, (1) shows how `rpcid`s 0.1.1 and 0.1.1.1 are added to the trace to fill the hole between the existing `rpcid`s.

Missing values captured in redundant rows: We use information captured in duplicate rows or through call paths to fill in missing `um`s and `dm`s.

Example: In figure 5.2, (2) shows a recovered (missing) `rpcid` 0.2.1, which was added to connect the existing

rpcids. In table format, we can refill 0.2.1's um with dm B (from its upstream call) and dm C (from its downstream call).

5.4.1.2 Context propagation errors (unique paths)

Context propagation errors (CPEs) can be fixed at the source: As stated in §**??**, a CPE originates from a service which incorrectly uses the same rpcid for different calls. We call a CPE's origin the source of the error. By differentiating each rpcid, we can always rebuild the trace structure at the source of a CPE.

First, we calculate the number of unique calls that were incorrectly assigned the same rpcid. The 2021 traces and 2022 traces use rts differently, so we handle each case independently. The 2021 traces use negative rts for reply rows. When the rt is below a certain threshold, it is rounded down to 0 (both for the call and reply edges). The call rt includes the child execution time plus the network latency while the reply rt is only the child execution time, so the reply edges must have a rt less than or equal to the call rt. We use these characteristics to calculate the number of calls for a given rpctype to each dm. For one-way communication, the rt values should all be positive (mostly 0), so the number of calls is equivalent to the number of rows.

For two-way communication in the 2021 dataset, we calculate number of calls as follows:

$$\begin{aligned} num\_fast\_calls &= \lfloor \frac{(rt == 0)}{2} \rfloor \\ extra\_fast\_row &= (rt == 0) \% 2 \\ num\_slow\_calls &= \max(-\texttt{rt}, abs(+\texttt{rt} - extra\_fast\_row)) \\ num\_calls\_to\_DM &= num\_fast\_calls + num\_slow\_calls \end{aligned} \tag{5.1}$$

-rt is the number of rows with negative response times and +rt is the number of rows with non-zero positive response times. Since two-way communication should have one positive and one negative row, where the negative row could be 0, we get the minimum number of unique calls if we maximize the number of +/- pairs made. num_fast_calls counts the pairs of rows with 0 rt. $extra\_fast\_row$ is 1 if there is an odd number of rows with 0 rt. We try to pair any leftover 0 rt rows with +rt rows. num_slow_calls tries to match the 0 rt row with a positive rt row (taking the absolute value when there are no positive rt rows), and then counts the pairs between the remaining + rt rows and - rt rows. Taking the max gives us the total number of pairs and the remaining unpaired rows. Finally, we add the fast and slow calls to get the total number.

The 2022 traces only have positive rts, the number of calls is calculated by:

$$num\_calls\_to\_DM = ceiling(rt/2) \tag{5.2}$$

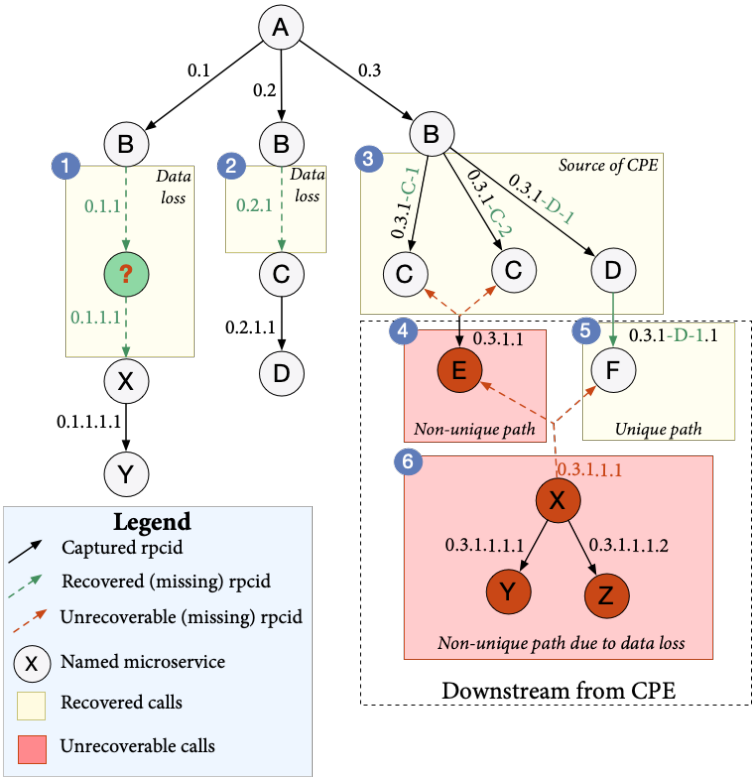


Figure 5.2: **CASPER example trace.** Regions of the trace that are corrected are highlighted in yellow and regions that are fundamentally unfixable are highlighted in red.

for two-way communication calls (and is the number of rows for one-way communication).

Since we can always be missing rows, the true number of unique calls is unknown. These calculations determine the minimum number of unique calls.

Using the minimum number of calls to each `dm`, we can fix the topology at the source of a CPE. We do this by updating the `rpcid`s to be unique for each call. We modify the `rpcid`s to be of the form $rpcid-DM-i$ where i is between 1 and the minimum number of calls to the `dm`.

Example: In figure 5.2, (3) shows how the `rpcid`s are updated to be unique for a CPE. The tabular version of this data (table 5.3) has five rows for the `rpcid` 0.3.1. We calculate that there are two calls to C and one to D. We update the `rpcid`s to be: 0.3.1-C-1, 0.3.1-C-2, and 0.3.1-D-1.

Unique call paths downstream from CPEs: When there is only one call to a `dm` at the source of a CPE, there is a possibility that we can rebuild the trace downstream from this service. If there are multiple calls to a `dm`, we cannot determine which instance of the `dm` made which downstream calls.

All downstream `rpcid`s from CPEs are not assumed to be unique because they share a non-unique `rpcid` in their ancestry. As a result, we can only rely on the call depth information from `rpcid`s as being accurate. We can use `um` and `dm` information to rebuild call paths downstream from CPEs when the call path is 1) unique (i.e.

the chain of `um`, `dm`, and call depth information forms a single valid call path) and 2) complete (there is no data loss or unknown values in the call path).

Example: In figure 5.2, (5) shows a unique path downstream from a CPE that we can connect to the trace. Table 5.4 shows the tabular version of this portion of the trace, where there is a row for 0.3.1.1 with `um` D, connecting to the single D node in the trace. We update the `rpcid` to be unique by changing its non-unique ancestor to be unique (e.g. 0.3.1.1 $\rightarrow$ 0.3.1-D-1.1).

5.4.2 Unrecoverable inconsistencies

5.4.2.1 Data loss

Data loss that has no redundancies: Most instances of data loss have redundancies in the dataset. However, there are some instances where there are no redundancies and the data is unrecoverable. For example, when sequential `rpcid`s are missing, not all `um` and `dm` information is recoverable.

Example: In figure 5.2, (1) shows that we are able to recover the missing `rpcid`s 0.1.1 and 0.1.1.1, but we cannot determine the service name for the additional node.

5.4.2.2 Context propagation errors (non-unique paths)

Non-unique call paths downstream from CPEs: We cannot remedy calls downstream from CPEs when they do not form unique call path. In the presence of data loss, it is fundamentally not possible to replace the missing calls since we cannot determine a unique ancestry to reconnect via. When we have missing `um` or `dm` values, they are not recoverable since there are often many unique possible values.

Example: In figure 5.2, (4) shows how the `rpcid` 0.3.1.1 (from table 5.3) cannot be uniquely connected to the trace. Since we cannot definitively connect this edge to the existing trace, we remove it and all rows downstream from it. Figure 5.2 (6), which visually represents the data in table 5.5, is affected by data loss. We are missing a row for the `rpcid` 0.3.1.1.1, which is needed to determine if the node X connects uniquely to the trace via the node F or non-uniquely to the trace via node E.

Conflicting paths, where the `um` does not connect upstream: As described in section 5.3.3, conflicting `um`s have a special case where they are not downstream from a CPE. In this case, there is only one service upstream. Any rows with `um`s that do not match the upstream's `dm` are dropped as they form invalid call paths.

5.4.3 casper algorithm

The CASPER algorithm performs a best case reconstruction of the trace topologies and guarantees that the edges in the resulting graphs are accurate. We keep track of the number of unrecoverable `rpcid`s that are omitted from

the traces. CASPER begins with general preprocessing of the data including filling missing values with duplicate rows. The meat of the program is in handling data loss and CPEs, which is outlined below.

At a high-level, CASPER performs a breadth first search (BFS) traversal over the `rpcid`s in each trace. When it identifies data loss upstream from a `rpcid`, it recursively fills in missing calls until it connects upstream. When it identifies a CPE, it fixes the error at the source and attempts to reconstruct the `rpcid`s downstream from the CPE (algorithm 2) before returning to the BFS traversal (algorithm 1).

Algorithm 1 CASPER algorithm for a single trace

```
1:  rpcids←BFS sorted rpcids for this trace
2:  root_rpcids←allroots
3:  for rpcid in rpcids do
4:      upstream_rpcid = rpcid.rsplit('.',1)[0];
5:      if Data loss then
6:          while upstream_rpcid not in trace do '
7:              Add edge for upstream_rpcid;
8:              upstream_rpcid← next upstream_rpcid;
9:          end while
10:     end if
11:     if Context propagation error then
12:         DMs← unique DMs for rpcid;
13:         for DM in DMs do
14:             min_calls_to_DM←max(R+,R−);
15:             for i in min_calls_to_DM do
16:                 Add edge for rpcid_DM_i;
17:             end for
18:         end for
19:         Rebuild downstream CP rpcids (Alg 2);
20:     end ifAdd edge for rpcid;
21: end for
```

For each trace, CASPER is initialized by sorting the `rpcid`s in BFS order and identifying all root `rpcid`s, which have no upstream `rpcid`s (alg 1, lines 1–2). It then iterates over the `rpcid`s in BFS order from each root and performs:

1. Check for data loss: if the `rpcid` is not a root, drop the trailing '`.`' and check for a *upstream_rpcid* (alg 1, lines 4–5).

   (a) If the *upstream_rpcid* does not exist, recursively add calls (with `um`/`dm` values when possible) until we connect to the trace (alg 1, lines 6–9).

rpcid	um	dm	rpctype	rt
0.3.1.1.1.1	X	Y	db	+
0.3.1.1.1.2	X	Z	db	+

Table 5.5: **Unrecoverable call paths downstream from CPE, affected by data loss.**

2. Check for CPE: Calculate the minimum number of calls made by this `rpcid`. If there are more than the expected number of rows (alg 1, line 11):
   (a) Extract the list of unique `dm`s called by the `um`. For each `dm`, calculate the number of calls to the `dm`. Create a unique `rpcid` for each call to the `dm` of the form `rpcid`=`rpcid`−`dm`−i where i ranges from 1 to the number of call (alg 1, lines 12–18).
   (b) Extract all `rpcid`s downstream from the CPE and rebuild independently. These rows have different assumptions (i.e. that the `rpcid` is not unique) so must be handled separately (alg 1, line 19).
3. Add edge to the trace, if no errors (alg 1, line 20)

Algorithm 2 describes how we remedy `rpcid`s downstream from CPEs. At a high-level, CASPER first identifies and removes subtrees in the trace that are downstream from data loss (i.e. disconnected subtrees). Next, CASPER performs call path validation, filtering out non-unique call paths. Algorithm 2 is initialized with only the `rpcid`s downstream from the source of a CPE, which are sorted in BFS order (alg 2, lines 1–2).

1. Check for downstream data loss: For each `rpcid`, check if it has a dangling tree below it. If the `rpcid` has no direct children, but has descendants (alg 2, lines 3–4):
   (a) Get the list of descendant `rpcid`s and delete all rows with these `rpcid`s (alg 2, line 5–6).
2. For each *row* that is downstream from the CPE and not affected by data loss, calculate the number of calls this row connects to upstream. This is done by generating the *parent*_`rpcid`, filtering the parent rows that connect to this row's `um`, and calculating the number of calls for those rows (alg 2, line 9–10):
   (a) If this row connects to a unique call path: update it's `rpcid` to be unique, by replacing its ancestry `rpcid` with its corrected *parent*_`rpcid` (alg 2, line 11–12).
   (b) If this row connects to a multiple call paths: delete the row and any connecting downstream call paths. (alg 2, line 13–14)

Algorithm 2 supports fixing sequential CPEs as long as the call paths are unique.

5.4.4 Limitations

Missing `rpcid`s before or after all recorded `rpcid`s: Missing `rpcid`s that are smaller than the smallest `rpcid` in the table or larger than the largest `rpcid` in the table. As mentioned in [129], root `rpcid`s can be anything (although it's most commonly 0 or 0.1). Additionally, we found there can be multiple roots in a single trace. When the root `rpcid` is larger than 0.1 (i.e. starting at a 'depth' of greater than 2), it could be the true root of the trace or it could be missing `rpcid`s before it.

Algorithm 2 CASPER rebuild calls downstream from CPE

```
1: rows ← rows with rpcids downstream from CPE;
2: rpcids ←BFS sorted rpcids downstream from CPE;
3: for rpcid in rpcids do
4:     if No child rpcid exists but exists descendents then
5:         decendent_rpcids ← rpcid.* from rpcids;
6:         delete rows for decendent_rpcids;
7:     end if
8: end for
9: for row in rows do
10:    num_connecting_calls ← num calls row connects upstream to;
11:    if num_connecting_calls == 1 then
12:        update rpcid in row;
13:    else
14:        delete row & connecting downstream rows;
15:    end if
16: end for
```

Corrupted values: We must make assumptions about the trace data that allow us to rebuild the topology. In addition to the assumptions provided by Alibaba, we assume `traceid`s are accurate. If `traceid`s (or any other fields for that matter), are corrupted in unpredictable ways, we cannot guarantee we will identify it and can remedy it.

5.5 Evaluation of casper

We seek to answer the following regarding CASPER's efficacy in circumventing inconsistencies in the Alibaba datasets.

Q1: How are traces generated by CASPER different from traces generated with other approaches? More specifically, what trace topological characteristics change using different reconstruction approaches?

Q2 How much do the recovery mechanisms in CASPER impact trace topologies?

Q3 How many additional complete traces does CASPER reconstruct compared to filtering out all traces with any inconsistencies?

The answers to *Q1* are a cautionary tale to researchers using the Alibaba trace dataset that reconstructing traces ignoring errors will lead to vastly skewed results that may impact the design or evaluation of their research artifacts. The answers to *Q2* evaluate the effectiveness of the recovery mechanisms in CASPER. The answer to *Q3* informs us how much the built-in side-channel redundancies in the Alibaba traces are able to help correct the inconsistencies without dropping communication calls or filtering entire traces.

For all of these analyses, we use a randomly sampled 10.8% (2,240,550) of the 2021 trace data and 1% (5,079,746) of the 2022 trace data. We use a lower sampling rate for the 2022 dataset because it contains many more traces than the 2021 version. We used six `r6.xlarge` instances to split the datasets as per `traceid` and one `c5a.23xlarge`

EC2 instance w/90 threads to run the CASPER algorithm and other construction approaches described in this section. The other construction methods are variants on the CASPER implementation and run inline with its execution.

5.5.1 Comparing construction methods

5.5.1.1 Methodology

We consider three alternative approaches to CASPER for constructing traces without deep knowledge of the inconsistencies in the datasets: `naive-rpcid`, `naive-accurate`, and `partial`. Figure 5.3 illustrates how the four approaches differ when reconstructing one example trace that has inconsistencies. We describe the alternative rebuild modes below.

`naive-rpcid`: keeps the first occurrence of a unique `rpcid` in the table and neither detects nor recovers any inconsistencies (similar to construction process in §??). Figure 5.3 shows the first row for each `rpcid` represented as a graph. The trace is disconnected since `rpcid` 0.1.1 is missing. Additionally, we are missing node E since it was not captured in the first row for its `rpcid`.

`naive-accurate`: detects inconsistencies and ignores the entire trace if an inconsistency is identified. Figure 5.3 shows once the missing `rpcid` 0.1.1 is detected, all rows for the trace are deleted.

`partial`: keeps calls in the traces that are not affected by an inconsistency. When an inconsistency is identified, the entire downstream call path is removed from the trace. `partial` traces preserve the accurate portions of traces. Figure 5.3 shows once the missing `rpcid` 0.1.1 is detected, all downstream rows are deleted.

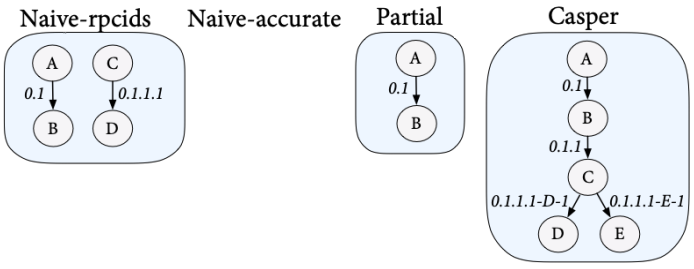
`CASPER`: as described in section **??**. Figure 5.3 shows how CASPER fixes the missing call 0.1.1 and the CPE at 0.1.1.1, updating the repeated `rpcid` to be unique for each call.

For `naive-accurate` and `partial`, we allow inconsistencies that are trivial to fix (e.g. missing values and missing duplicate rows) since they do not affect the trace topology.

To compare the four approaches, we analyze the following topological characteristics of the reconstructed traces: trace size, call depth, and width. Trace size represents the total number of microservices in the trace. A microservice

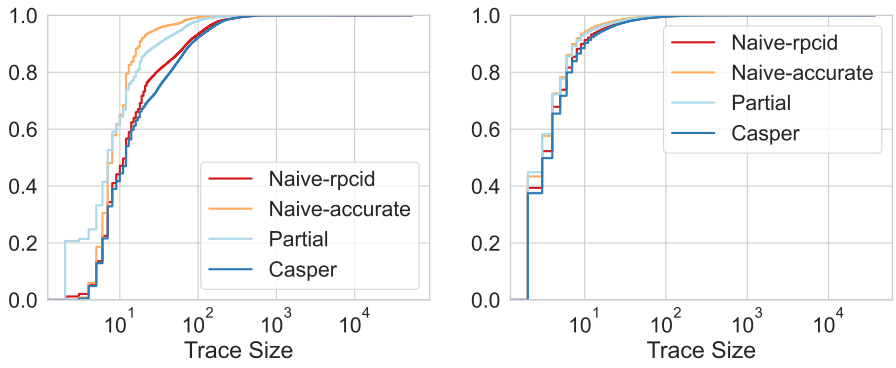
rpcid	um	dm	rpctype	rt
0.1	A	B	rpc	+/-
0.1.1.1	C	D	rpc	+/-
0.1.1.1	C	E	db	+

(a) Trace Example



(b) Modes

Figure 5.3: **Trace building modes.** Four different modes for building trace graphs, each has a different method of handling errors and inconsistencies.



(a) 2021 (b) 2022

Figure 5.4: Trace sizes for different modes

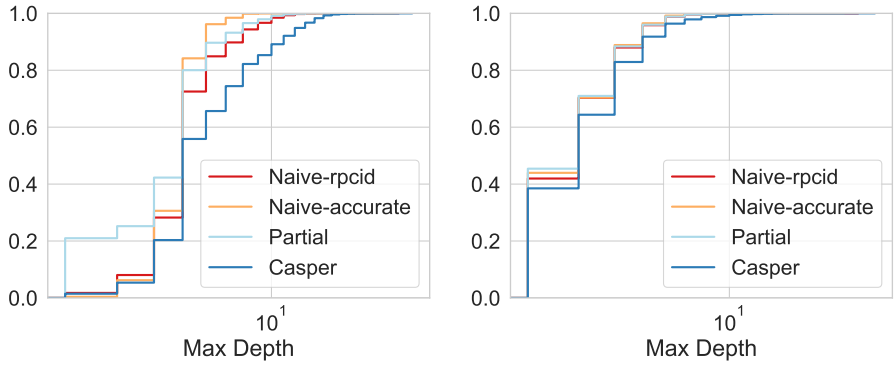
can be called many times within a trace and each call is included in the trace size. Call depth is the maximum depth of the call paths in the traces. Width is the maximum number of calls made by a single microservices. In graph form, this is the largest fan-out. We compare the cumulative distribution functions (CDFs) for all metrics.

5.5.1.2 Results

Overall, we find that CASPER traces are larger, wider, and deeper than all other methods of constructing traces for both the 2021 and 2022 datasets. The 2022 traces are smaller (size, depth, and width) than the 2021 traces. The 2022 traces have less inconsistencies, so CASPER's impact on the trace topology compared to `naive-rpcid` is less significant.

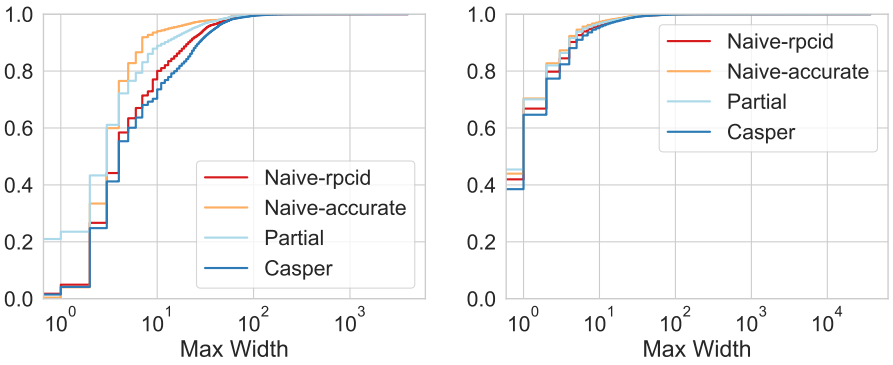
Trace size: For both the 2021 and 2022, CASPER builds larger traces than all other approaches by correcting data loss and CPEs. Figure 5.4 shows the CDF of trace sizes for both years. For 2021, at the 50th percentile (P50), a CASPER trace is the same size as `naive-rpcid` traces. However, CASPER produces larger traces at P75 than all other modes. On average, CASPER traces have size 33.08 whereas the size is 12.22, 15.50, and 29.11 for `naive-accurate`, `partial` and `naive-rpcid` respectively. For 2022, CASPER traces have similar, the same or slightly bigger, size compared to all other modes at all percentiles. Traces from 2022 are overall smaller than those from 2021. The average trace size `naive-accurate` is 12.22 in 2021, but is decreased to 4.98 in 2022.

Call depth: For both the 2021 and 2022, CASPER builds deeper traces than all other approaches by reconnecting call paths that have missing `rpcids`. Figure 5.5 shows the CDF of the maximum call depth of a trace for both years. For 2021, at P50, CASPER traces have the same depth as the other modes. However, CASPER produces deeper traces at P75 than all other modes. On average, CASPER traces have depth 6.40 whereas the depth is 4.84, 4.56, and 5.26 for `naive-accurate`, `partial` and `naive-rpcid` respectively. For 2022, CASPER traces have the same depth or slightly deeper than all other modes at all percentiles. Traces from 2022 are overall shallower than those from 2021. The average depth for a `naive-accurate` trace is 4.84 in 2021, but is decreased to 3.01 in 2022.



(a) 2021 (b) 2022

Figure 5.5: Maximum trace depth for different modes



(a) 2021 (b) 2022

Figure 5.6: Maximum trace width for different modes

Width: For both the 2021 and 2022, CASPER builds wider traces than all other approaches by differentiating repeated `rpcid`s generated by CPEs. Figure 5.6 shows the CDF of maximum width of a trace for both years. For 2021, P25 and P50, a CASPER trace has similar width as the other modes. At P75, CASPER traces are wider. On average, CASPER traces have width 10.73 whereas the width is 5.30, 5.91, and 8.97 for `naive-accurate`, `partial` and `naive-rpcid` respectively. For 2022, CASPER traces are similarly wide or slightly wider than all other modes at all percentiles. Traces from 2022 are overall narrower than those from 2021. The average width for `naive-accurate` traces is 5.30 in 2021, but is decreased to 1.92 in 2022.

5.5.2 Impact of recovery mechanisms

5.5.2.1 Methodology

We break down CASPER's recovery mechanisms (described in §5.4.3) into four parts and quantify their impact. 1) *Adds missing calls* counts the number of new calls added to a trace. 2) *Fills in missing values* counts the originally unknown microservice names that CASPER recovers. 3) *Updates `rpcid`s at a CPE source* measures the number

of `rpcid`s added when differentiating calls at the first occurrence of a CPE in a call path. 4) *Recovers `rpcid`s downstream from a CPE source* counts the number of calls CASPER identified as uniquely connected to the trace downstream from the first CPE in the call path.

CASPER collects metrics for each of the recovery mechanisms when reconstructing the traces. We calculate the number of traces that are affected by each recovery and its impact within the trace.

5.5.2.2 Results

Adds missing calls: CASPER recovers all missing internal calls in a trace. For 2021, 30.47% of traces have at least one missing internal `rpcid`, with the average number of calls 10.7 (std: 23.79, P99: 115). For 2022, 8.77% of traces have at least one missing call added by CASPER, with average 3.57 (std: 13.12, P99: 30). CASPER reconnects broken traces, resulting in longer call paths.

Fills in missing values: CASPER fills in missing `dm` values using duplicate information stored in rows or call paths. For 2021 traces, we are able to recover at least one `dm` in 99.98% of traces. On average, traces have 2.84 recovered `dm` names (std: 5.61, P99: 27). For 2022 traces, we are able to recover at least one `dm` in all traces. On average, traces have 2.61 recovered `dm` names (std: 6.99, P99: 34).

Updates `rpcid`s at a CPE source: Given a call path starting from the root call, the first occurrence of a CPE is a CPE source. CASPER modifies `rpcid`s to be unique to differentiate different calls that originally shared the same `rpcid`. This modification may preserve more branches and yield wider traces. For 2021 traces, CASPER modifies on average 2.77 `rpcid`s per CPE source (std: 2.65 and P99: 11). For 2022 traces, CASPER modifies on average 2.02 `rpcid`s at a CPE source (std: 0.18 and P99: 3).

Recovers `rpcid`s downstream from a CPE source: CASPER connects unique call paths downstream from a CPE source, updating their `rpcid`s to be unique which may preserve longer call path and yields deeper traces. We measure this impact by counting the number of such updated `rpcid`s per CPE source. For 2021 traces, CASPER modified on average 3.08 downstream `rpcid`s (std: 11.22 and P99: 48). For 2022 traces, CASPER modified on average 1.11 downstream `rpcid`s (std: 32.97 and P99: 19). Note that additional CPEs can occur downstream, but they are rare. For 2021, the average number downstream CPEs is 0.23, (std: 1.5 and P99: 6). For 2022, the average number downstream CPEs is 0.14, (std: 2.71 and P99: 3).

5.5.3 Additional complete traces

5.5.3.1 Methodology

We evaluate CASPER's effectiveness at rebuilding complete traces by measuring the number of additional complete traces output by CASPER (when compared to `naive-accurate`). A trace is complete if there are no unrecoverable

`rpcid`s (explained in Section 5.4.3).

5.5.3.2 Results

For 2021, 58.32% of the traces have complete topology without needing to remedy any inconsistencies. CASPER reconstructs an additional 25.5% of the traces, totaling to 83.82%.

For 2022, 86.42% of the traces have complete topology without needing to remedy any inconsistencies. CASPER reconstructs an additional 12.18% of the traces, totaling to 98.6%.

5.6 Discussion

Recommendation for consumers of the Alibaba datasets and related research papers: Users of the datasets should always specify their methodology for identifying and mitigating inconsistencies in their analyses. They should prefer the 2022 dataset, which contains fewer total inconsistencies. But, it is unclear if traces in the 2022 dataset were collected from the same applications or application versions as the 2021 dataset. The maximum trace sizes and maximum widths differ significantly between both years regardless of rebuild mode. As such, consumers may wish to use both datasets to test their work against a range of request-workflow characteristics.

We recommend caution when interpreting research that uses the 2021 dataset without specifying a methodology for handling inconsistencies. Readers should carefully consider if changes in trace characteristics or connectivity would affect the results. Of particular note is the Alibaba microservice analysis by Luo et al. [2]. This analysis uses a 7-day dataset of which the 2021 public release is a subset. But, does not specify whether the authors knew about the inconsistencies or whether they addressed them. As such, the presented graphs of trace characteristics, clustering results, and distributions suggest for the artificial trace generator may be suspect.

Exploring tradeoffs in capturing redundancies within trace data: CASPER's functionality is possible because Alibaba's tracing infrastructure stores redundant data in caller/callee log messages. Namely, `rpcid` uniquely locates a call in the trace, allowing call-graph connectivity between services when intermediate calls are lost. But, `rpcid`s' expressiveness results in larger context and larger network message sizes. Context-propagation errors can be circumvented by using chains of `um` / `dm` fields. In contrast, the popular open-source model for distributed-tracing, OpenTelemetry [14], does not capture any redundancies. Spans (e.g., service executions) can be dropped if services' are too resource starved [130], leading to traces with (silent) missing nodes. Research is needed to explore how to encode redundancies in trace data or context and the overhead tradeoffs of doing so.

Chapter 6

Bridging Gaps in Traces for Microservices

6.1 Overview

Distributed tracing promises end-to-end visibility into request workflows, but this promise breaks down when data is incomplete. Our analyses of production traces from Alibaba, Uber, and Meta reveal that 5–40% of traces contain *holes*—missing records dropped by overloaded infrastructure. Tools that assume complete traces may produce incorrect results when applied to fragmented data.

In this chapter, we introduce *bridges*, a new distributed tracing primitive that preserves trace characteristics in the presence of data loss. The key insight is that important trace properties can be encoded in compact *breadcrumbs* and propagated with requests, providing redundancy that enables reconstruction when records are dropped. We present three bridge types with different fidelity-overhead tradeoffs: Structural Bridges preserve both topology and sibling ordering for critical-path extraction, Call-Graph Bridges preserve topology with reduced overhead, and Path Bridges preserve only connectivity for minimal cost.

Section **??** motivates the problem with examples of data loss in production systems. Section **??** defines the trace characteristics we aim to preserve. Section **??** presents the three bridge types and their reconstruction algorithms. Section **??** describes Manchac, our implementation atop OpenTelemetry. Section 6.5 presents preliminary evaluation of overhead.

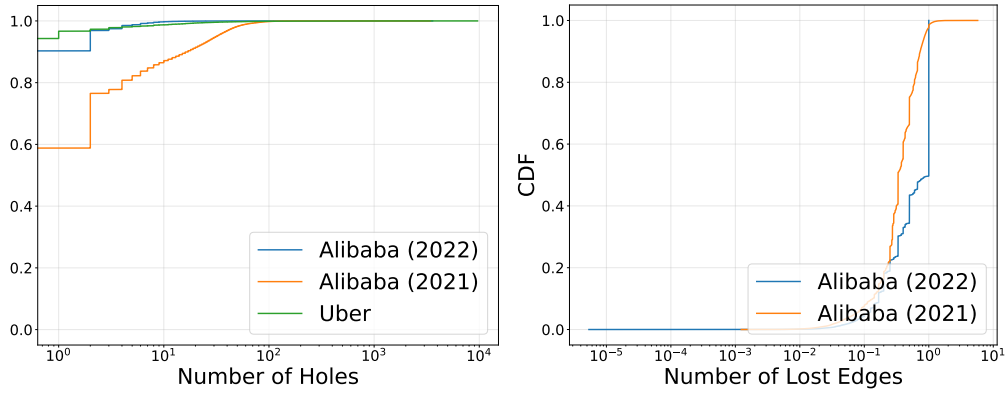
This work is ongoing, with submission planned for Spring 2026. The evaluation presented here focuses on overhead; full evaluation of reconstruction accuracy and impact on downstream tools is in progress.

6.2 Why do we need bridges?

We build the case for bridges by analyzing characteristics of holes in distributed-trace datasets released by Alibaba [2], Meta [3], and Uber [**?**] (§6.2.2-§**??**). For our analysis, we define a hole as a sequential path of one or more missing trace records. We survey existing tracing-based management tools to identify which ones are affected by holes and conclude with a discussion of how bridges can help. We focus on Alibaba's datasets as their custom trace format enables more sophisticated analyses than the formats used by the other organizations. Our analyses assume that distributed tracing infrastructures are implemented correctly.

	Alibaba'21	Alibaba'22	Uber	Meta
Holes	41.21%	9.72%	5.72%	N/A
Lost edges	98.30%	80.10%	N/A	N/A
Total	1,596,938	40,130,255	531,085	

Table 6.1: **Traces w/lost trace records.** For Uber, traces with holes are ones that contain at least one dangling child span pointer. For Meta, ????. For Alibaba, we conservativley define traces with holes as two-way messages where both rows have been lost. We also plot the percentage of traces with missing edges, which may represent the single row captured for one-way messages or one of two rows of two-way messages.



(a) Holes per trace

(b) Lost edges per trace

Figure 6.1: **Holes or lost edges per trace.** Treating each hole as a path, which may be overlapping with other paths in adjacent holes.

This work is scoped to microservice applications using distributed tracing. See Section 6.2 for a comprehensive discussion of microservices and distributed tracing.

6.2.1 Data loss in distributed tracing

When can records be lost, resulting in holes? Distributed tracing infrastructures may drop trace records due to overloads or failures. With regards to overload, local caches within agents may run out of capacity if corresponding services' request arrival rate exceeds the rate at which caches can be emptied. Caches at collectors may also run out of capacity if the span-arrival rate is higher than the ingest rate. With regard to failures, trace-records may be lost if tracing agents or collectors fail.

When can baggage be lost? A key insight of this paper is that baggage and requests' propagation share the same fate. This means data in baggage can only be lost if services fail or encounter error conditions that terminate request processing itself. Such errors will correctly result in traces no longer being collected after the failure point and so cannot result in holes.

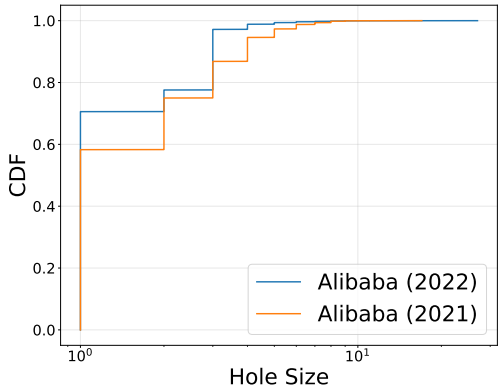


Figure 6.2: **Most holes are small.** Treating each hole as a path, which may be overlapping with other paths in adjacent holes.

6.2.2 Analyzing trace-record loss

We analyze trace-record loss across four datasets: one hour (hour 0) of Alibaba's 2021 dataset [**?**], one hour of day 1 of Alibaba's 2022 dataset [**?**], all of Uber's dataset [**?**], and Meta's reported loss rates [**?**].

The datasets use different trace formats. Uber uses OpenTelemetry [14], where each span record links to its parent via propagated context; trace records correspond to span executions. Alibaba uses a custom format where each recorded call preserves the full caller chain from root to current service; trace records represent edges between parents and children. For two-way communication (e.g., RPCs), two records capture each call: one at the caller, one at the callee. For one-way communication (e.g., database calls), a single record at the caller is recorded.

A significant fraction of traces contain holes: Table **??** summarizes trace-record loss across datasets. In Alibaba's 2021 dataset, over 40% of traces contain at least one hole, and 98% are missing at least one edge record. These rates drop to 9.7% and 80.1% respectively in the 2022 dataset. For Uber, 5.7% of traces contain holes (lost edges are not applicable since records represent span executions, not edges). The research community has asked Alibaba why records are lost [**?**], but no definitive answer has been provided. The cause of loss in Uber's traces is also unknown.

Traces with holes typically contain multiple holes: Figure **??** shows the distribution of holes per trace. When a trace contains a hole, it often contains several. Across traces with at least one hole, the 99th percentile hole count is **MISSING_VAL 1** (Alibaba 2021), **MISSING_VAL 2** (Alibaba 2022), and **MISSING_VAL 3** (Uber).

Most holes are small: Alibaba's format, which propagates the full ancestry chain, allows us to measure hole sizes in terms of missing calls (Figure **??**). In the 2021 dataset, 70% of holes contain only one missing call (**MISSING_VAL 4**% in 2022). The 95th percentile hole size is **MISSING_VAL 5** calls (2022: **MISSING_VAL 6**), with a maximum of **MISSING_VAL 7** (2022: **MISSING_VAL 8**).

Upsampling complete traces does not recover lost information: One might hope that complete traces

Name	Trace Record Names	Call graph connectivity	Ordering
		Auto-scaling	
FIRM [99]	Service, Endpoint, Instance	Yes	Yes, for critical Path
Sage [98]			
Sinan [100]	Service, Endpoint	Yes	No
		Sampling	
Sifter [131]	Service, Endpoint	Yes	No
Sieve [**?**]	Service, Endpoint	Yes	Yes
STEAM [**?**]	TODO	Yes	Depends on domain specific information
		Performance debugging	
Spectroscope [108]			
tprof [**?**]	root endpoint, service, endpoint	Yes	Yes, Level 4
CRISP [102]	Service, Endpoint	Yes	Yes, for Critical Path

Table 6.2: **Papers in top systems conferences, tracing features they use, and the impact on them of lost data.**

could be upsampled to compensate for holes. However, this assumes that a parent's children are deterministic—i.e., that observing one complete execution of a parent suffices to know what children it will call in all executions. We tested this assumption by measuring how often parents call the same set of children across invocations.

We define a *children set* as the set of endpoints a parent calls in a single invocation. A parent is *deterministic* if it always calls the same children set, or if its children sets are mutually exclusive (indicating distinct code paths that could be inferred from trace data). Otherwise, when children sets overlap but vary, we cannot reliably infer which children are missing in a hole. We found that **MISSING_VAL 9**% of parents are non-deterministic: they call varying children sets that cannot be distinguished from available trace data. This means upsampling cannot reliably reconstruct what was lost.

6.2.3 Trace-record loss's impact on management tools

Distributed traces are the foundation for a wide range of microservice management tools including auto-scalers, anomaly detectors, and performance debuggers. These tools assume traces accurately represent request workflows, an assumption potentially violated when trace records are lost. Table 6.2 summarizes key research systems that rely on trace data and the trace features they require.

Impact on trace-based tools: Trace-record loss affects different tools in different ways depending on which trace features they use:

- **Grouping and aggregation:** Tools that group traces by structure such as Sifter [131] for sampling or Spectroscope [108] for performance debugging may incorrectly classify incomplete traces as distinct from their complete counterparts, skewing aggregate statistics.

- **Critical path extraction:** Auto-scalers like FIRM [99] and debuggers like tprof [**?**] extract critical paths

to reduce data ingest by focusing on nodes impacting end-to-end latency. Missing spans break the path, causing these tools to either fail or incorrectly attribute impact.

- **Topology-based analysis:** Tools like Sinan [100] learn models based on call-graph structure. Holes in traces produce incomplete or disconnected graphs, potentially degrading model accuracy.
- **Anomaly detection:** Systems that detect anomalies by comparing trace structure or latency distributions may flag incomplete traces as anomalous, generating false positives, or miss true anomalies obscured by data loss.

6.2.4 Requirements for bridging mechanisms

A prerequisite to addressing data loss in traces is the ability to *detect* it. Traces must provide a systematic way to indicate when spans are missing and where gaps occur. At a minimum, a system must:

R0 **Flag missing spans or trace points**: provide a clear signal that data loss has occurred in the trace.

Detection alone is insufficient. We argue for the notion of *holes*—first-class markers of data loss that explicitly preserve connectivity between the observed and missing parts of a trace. A system that supports holes should:

R1 **Preserve connectivity**: When data is lost in a trace, there should be a connection between the lost data and the remainder of the trace. This explicitly illustrates that data loss exists and the location of the data loss.

R2 **Minimal overhead**: The solution should add minimal overhead to the application, tracing framework, and developer.

R3 **Require little to no modifications from developers**: The solution should be able to be deployed without modifying the application.

6.3 Different types of bridges

This section presents three bridge primitives that preserve different trace properties in the presence of data loss. Each bridge type supports different downstream use cases (as summarized in Table 6.2) and incurs different overhead.

The **Strucural Bridge** (§6.3.2) preserves call-graph topology and ordering relationships between sibling calls, supporting critical-path extraction and full trace reconstruction. The **Call-graph Bridge** (§6.3.3) preserves call-graph topology without ordering, supporting trace visualization and topology-based analysis. The **Path Bridge** (§**??**) preserves only transitive connectivity between fragments, supporting some grouping and aggregation use cases.

Each relaxation of requirements reduces the complexity of the metadata that must be propagated, moving along

the trade-off space between reconstruction fidelity and runtime overhead. All three bridge types are compatible with span-based tracing infrastructures that use one-way context propagation (e.g. OpenTelemetry [14]). They require only modifications to tracing SDKs and custom collector plugins; no changes to application business logic are needed. Implementation details are discussed in §6.4.

6.3.1 Key Ideas

Bridges work by embedding metadata in request context describing previous trace characteristics. We call this metadata *breadcrumbs*. Because breadcrumbs are propagated in-band with requests, they can only be lost if the request fails, in which case the trace should correctly terminate.

Key idea 1: Breadcrumbs: Trace characteristics such as caller/callee relationships, call-graph structure, and sibling ordering can be encoded in compact data structures small enough to propagate with trace context. Table 6.3 summarizes the key-value pairs stored by each bridge type. *Fingerprint* is used to describe a unique identifier for a span (e.g. *spanID*). Only the attributes needed for the specific bridge type are stored in the breadcrumb. Breadcrumbs are added to baggage and travel with requests through the system, accumulating ancestry information as they pass through services. When trace records are lost, the breadcrumbs in surviving records contain the information needed for reconstruction.

Key	Value	S	CGP	Path
`depth`	`int`: relative call depth	✓	✓	✓
`ancestors`	`[fp, ...]`	✓		
`AMQ(ancestors)`	`AMQ filter of ancestor fingerprints`		✓	✓
`forks` (optional)	`[fp:depth, ...]`		✓	
`lateral` (optional)	{`parent_fp: [fp:ordinal, ...], ...`}	✓		
`ordinal` (optional)	*int*: ordinal number	✓		
`delayed_lateral` (optional)	`(trace_id:lateral_breadcrumbs)`	✓		

Table 6.3: Breadcrumb key-value pairs by bridge type. "FP" denotes fingerprint.

Key idea 2: Vertical ancestry: The foundation of bridge reconstruction is preserving parent-child relationships across arbitrary depths. *Vertical ancestry* is breadcrumb data propagated along a single path of a request's execution, updated by each service before calling downstream services. This data accumulates information about all ancestors from the root to the current service, enabling reconstruction of connectivity even when intermediate spans are lost.

Key idea 3: Lateral ancestry: Call-graph topology alone is insufficient for use cases like critical-path extraction, which require knowing the ordering between sibling calls. *Lateral ancestry* capture relationships between children called by the same parent. Unlike vertical ancestry, lateral ancestry is generated by a single service instance

Bridge Type	Description
	Exact structure
Structural Bridge	preserve both ordering and topology
	Exact hierarchy
CGP Bridge	preserves topology, reduces overhead when high concurrency
Exact Bridge	preserve exact topology and node fingerprints
	Lossy hierarchy
Path Bridge	preserve connectivity, cannot record lost topology

Table 6.4: **Bridge types.** different types of bridges

after observing the responses from its children. It records which children were called and their relative ending orders (sequential vs. concurrent).

Because some lateral ancestry is generated after children respond, the parent may not have any additional calls to attach the metadata to. To handle this, lateral ancestry can be attached to a *future* request passing through the same service instance.

6.3.2 Structural Bridge (S-Bridge)

The Structural Bridge enables critical path extraction and performance analysis even when intermediate spans are lost. Critical path analysis identifies the sequence of operations that determined a request's end-to-end latency, which requires knowing whether sibling calls were sequential or concurrent and in what order they completed.

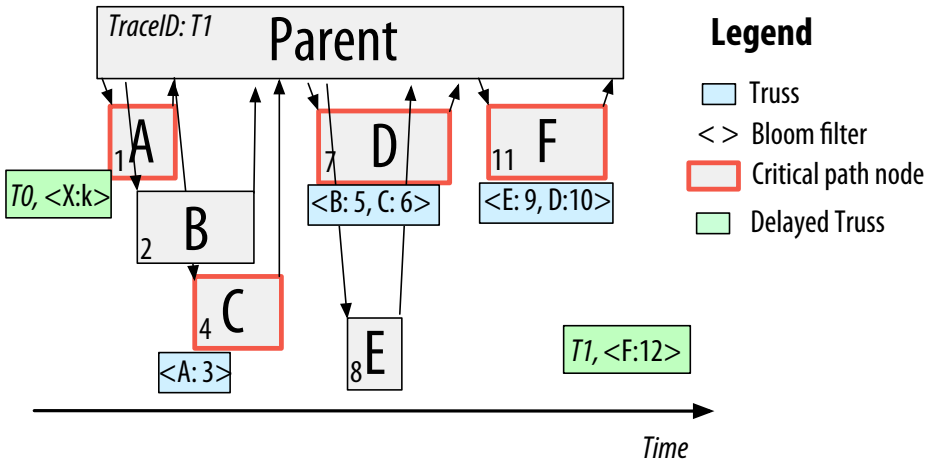


Figure 6.3: **Structural Bridge.** Shows the trusses generated to preserve ordering relationships among sibling calls.

Breadcrumbs: The S-Bridge generates two types of relational data (Table 6.3). Vertical breadcrumbs preserve parent-child relationships by propagating a list of ancestor fingerprints in baggage. Each service appends its own fingerprint before calling downstream.

Algorithm 3 Minimal-State Ordinal Annotation for Sibling Ordering

Parent state:

1: $k \leftarrow 0$ ▷ Monotonic counter local to parent span
2: $trusses \leftarrow [\,]$
3: **if** Child A starts **then**
4: $k \leftarrow k+1$
5: SetAttr(`child_start_ord`, k)
6: Propagate(`delayed_trusses, trusses`)
7: $trusses \leftarrow []$
8: **end if**
9: **if** Child A ends **then**
10: $k \leftarrow k+1$
11: $trusses \leftarrow (A,k)$
12: **end if**
13: $delayed\_trusses \leftarrow (trace\_id, trusses)$

Lateral breadcrumbs capture sibling ordering using ordinal numbers. As shown in Algorithm 3, the parent maintains a counter k that increments with each child call and response (line 3 and 9). Each child is represented by two events: a *start event* when the parent issues the call, and an *end event* when the parent receives the response. Each event receives an ordinal number. Start ordinals are added to the child's baggage immediately (line 5) and added to the breadcrumb for that child (as child's fingerprint: *start_ordinal*). End ordinals are locally stored as *(start_ordinal, end_ordinal)* pairs and passed to the next child call (lines 11 and 6). Storing the corresponding start ordinal for an end ordinal allows us to attribute the end ordinal to a specific child span. The child's span ID is unknown to the parent, so it can't be used here. The counter uses atomic increment operations to avoid lock contention.

After all children complete, any remaining end ordinals become *delayed breadcrumbs*. As illustrated in Figure 6.3 (green annotations), delayed breadcrumbs include the trace ID and attach to the next outgoing call from the parent service instance, potentially on a different trace. This ensures ordering information survives even if the parent's span is dropped.

.

Reconstruction: Reconstructing lost spans requires recovering their parent-child relationships and the ordering between siblings at fan-out points. The algorithm must maintain three invariants:

1. **Containment**: A span must be fully contained within its parent: $time(P_s) \leq time(A_s) \leq time(A_e) \leq time(P_e)$
2. **Encompassing**: A span's timestamps must encompass all children: $time(A_s) \leq \min(time(c_s))$ and $time(A_e) \geq \max(time(c_e))$ for all children c
3. **Ordinal consistency**: Reconstructed ordinals must match those captured in breadcrumbs

Reconstruction proceeds in two phases (Algorithm 4):

Phase 1: Topology reconstruction. The call graph is rebuilt from vertical breadcrumbs, which contain the exact fingerprints of all ancestors. This phase is straightforward since fingerprints uniquely identify each span. For example, if a disconnected span D has breadcrumb [A,B,C,D] then it must be the last span on a path from A, to B, to C, to D.

Phase 2: Ordering reconstruction. For spans without timestamp information (e.g. originally lost, but reconstructed during phase 1), we first set initial timestamps to tightly encompass each node's children (lines 1–6). Sequential siblings remain correctly ordered after this step since their children are also sequential. For concurrent siblings that appear sequential after tight coupling, we shift both timestamps by $d/2+\epsilon$ toward each other, where d is the time gap between them and ϵ is **MISSING_VAL 10** (lines 13–23). Figure 6.4 illustrates how this adjustment affects the critical path.

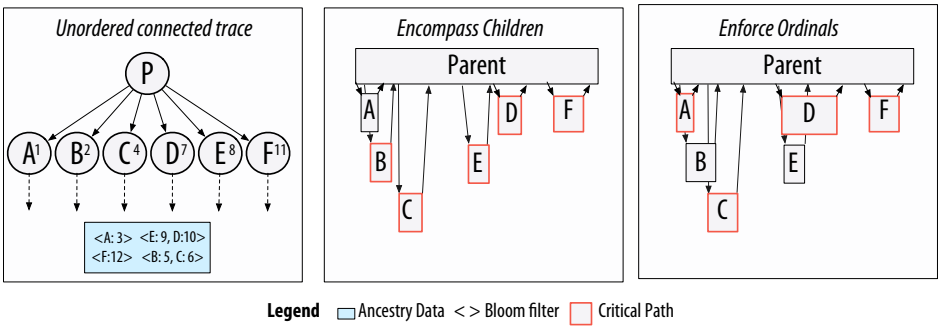


Figure 6.4: **Structural-Bridge Reconstruction.** Assumes all nodes have children, but omitted for simplicity

The reconstructed timestamps assigned to each span are estimates and actual span durations may differ. However, the ordering relationships are preserved, enabling accurate critical-path calculation.

Overhead: Breadcrumb generation adds minimal latency: ordinals use atomic increments (avoiding mutex contention). For high fan-out services, fingerprint:ordinal pairs can be compressed into AMQ filters, which are queried during reconstruction by checking likely filters first (starting with the nearest filter after the start event). Reconstruction runs in time proportional to the number of lost spans and their siblings.

6.3.3 Call-graph Bridge (CG-Bridge)

The Call-Graph Bridge preserves trace topology without capturing sibling ordering. By omitting lateral breadcrumbs, the CG-Bridge reduces in-band overhead compared to the S-Bridge's topology preserving breadcrumbs.

Breadcrumbs: The CG-Bridge propagates only vertical breadcrumbs (Table 6.3). As shown in Figure 6.5, each span is annotated with an AMQ filter containing the fingerprints of all ancestors on its call path. This enables reconstruction of parent-child connectivity when intermediate spans are lost: we can query the filter for containment of the collected span's fingerprint to find its nearest ancestor.

Algorithm 4 Reconstruction of structural bridges

function BUILDCHILDBRIDGES(P) ▷ Encompass children timestamps (initialize baselines)
  for all $s \in \mathcal{S}$ **do**
    $children \leftarrow s.\texttt{children}$
    $s_s \leftarrow \min_{c \in children} c_s$
    $s_e \leftarrow \max_{c \in children} c_e$
  end for
  ▷ Enforce ordinal concurrency
  for all pairs (A,B) in $\mathcal{S}$ **do**
    if $\neg(ord(A_e) < ord(B_s)) \wedge \neg(ord(B_e) < ord(A_s))$ **then**
      ENFORCECONCURRENT(A,B,P)
    end if
  end for
end function

function ENFORCECONCURRENT(A,B,P) ▷ If baseline already overlaps, do nothing
  if $A_e < B_s$ **then**
    $d \leftarrow B_s - A_e$
    $A_e \leftarrow \min(A_e + d/2 + \epsilon,\ time(P_e))$
    $B_s \leftarrow \max(B_s - d/2 - \epsilon,\ time(P_s))$
  else if $B_e < A_s$ **then**
    $d \leftarrow A_s - B_e$
    $B_e \leftarrow \min(B_e + d/2 + \epsilon,\ time(P_e))$
    $A_s \leftarrow \max(A_s - d/2 - \epsilon,\ time(P_s))$
  end if
end function

The key challenge is preserving fan-out points. Without additional information, reconstruction cannot distinguish whether two disconnected spans share a common (lost) parent or descend from separate ancestors. This is because we can only query the AMQ filter for spans that are not lost, as we do not know the fingerprints of lost spans. To address this, the *second* child issued from any fan-out carries an additional breadcrumb: an array of *(fingerprint, depth)* pairs identifying the fan-out point and what depth it occured at. Storing this only on the second child ensures fan-out information exists on exactly one subtree, minimizing redundancy. For example, in Figure 6.5, record D carries the fan-out breadcrumb for B's fan-out.

.

Reconstruction: Algorithm 1 reconstructs the call-graph topology in two phases:

Phase 1: Establishing connectivity. For each disconnected span, we identify its nearest collected ancestor by querying AMQ filters. The depth difference between the two spans indicates how many intermediate spans were lost. We insert *synthetic spans* to bridge the gap, connecting them with bridge edges (Figure 6.6, step 1).

Phase 2: Reconstructing fan-outs. Synthetic spans may represent the same lost fan-out point duplicated across multiple subpaths. Using the *(fingerprint, depth)* arrays, we first mark all synthetic spans that correspond to known fan-outs. We then traverse the graph breadth-first, processing each depth level in turn. At each depth, we check whether any synthetic spans are known fan-out points. For each fan-out found, we identify merge candidates:

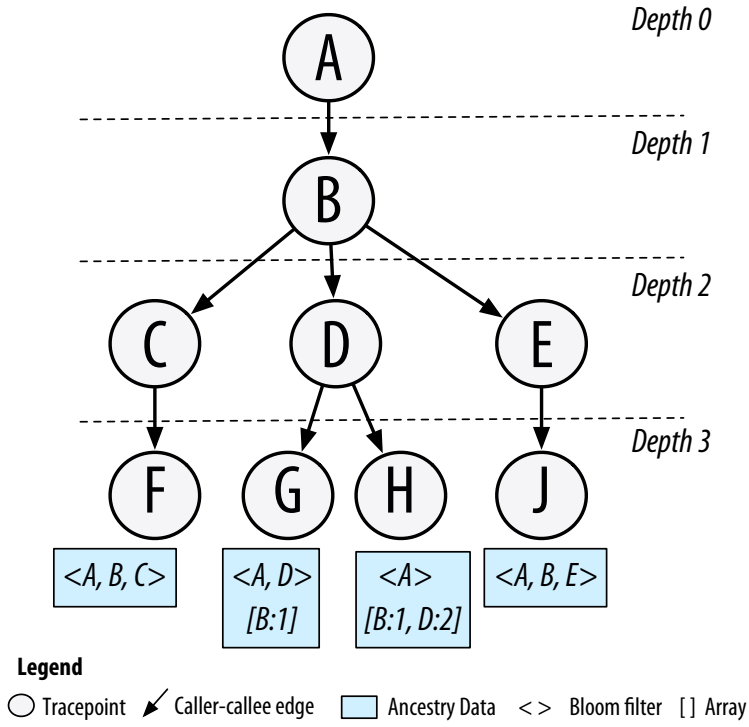


Figure 6.5: **CGP-Bridge.** The structure of a CGP-Bridge. Ancestry data is only shown on leaves for simplicity.

synthetic spans with the same parent as the fan-out. We query each candidate's AMQ filter for the fan-out fingerprint; if present, the candidate is a child of the fan-out and its synthetic parent should be merged with the fan-out span. This continues until all synthetic spans have been processed (Figure 6.6, step 2).

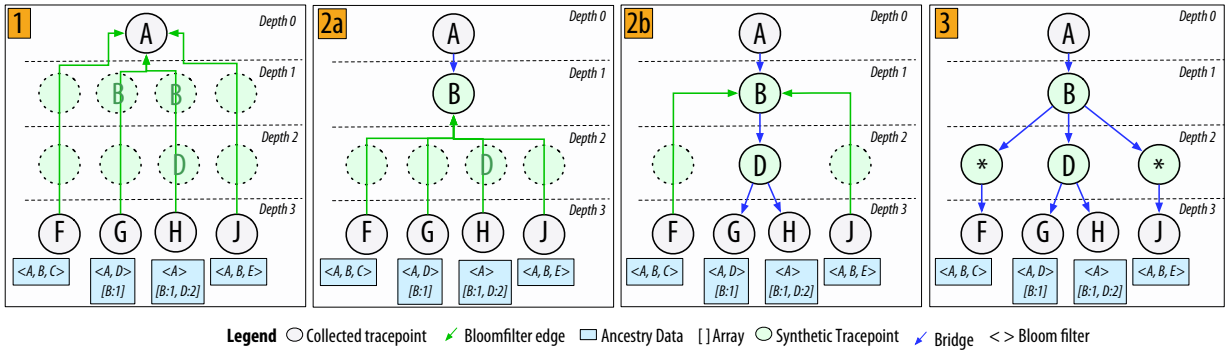


Figure 6.6: **CGP-Bridge Reconstruction.** Rebuilding missing tracepoints using ancestry data. Example shows all non-root and non-leaf tracepoints initally are missing.

The resulting graph has the same topology as the original trace.

Overhead: Breadcrumb generation adds minimal latency. AMQ filters are updated incrementally as requests propagate. Breadcrumb size scales with call depth and fan-out frequency. In the worst case (deep traces with frequent fan-outs) the second child at each fan-out carries an array proportional to the number of upstream fan-outs.

Algorithm Reconstruction of CGP-Bridges

```
1: procedure ESTABLISH_CONNECTIVITY(dnodes)
2:   for n∈dnodes do
3:     ancestory←nearest_ancestor(n)
4:     missing_nodes←this_depth−ancestor_depth−1
5:     add missing_nodes to nodes
6:   end for
7: end procedure

8: procedure MERGE_FANOUTS(nodes)
9:   for d∈max_depth do
10:    depth_nodes←nodes_at_depth(d)
11:    known_fanouts←depth_nodes.filter(is_known_fanout)
12:    for k∈known_fanouts do
13:      candidates←depth_nodes.filter(parent==k.parent)
14:      merge_nodes←candidates.filter(bloomfilter.contains(k))
15:      remove all but one candidate node and update edges.
16:    end for
17:  end for
18: end procedure
```

However, this remains substantially smaller than S-Bridge overhead since no lateral ordering data is captured and all fingerprints that aren't explicitly stored as a fan-out are compressed in AMQ filters. .

6.3.4 Path Bridge (P-Bridge)

The Path Bridge is the lightest of the three bridge types, designed for use cases that only require reconnecting disconnected trace fragments. When spans are lost along a call path, downstream fragments become orphaned. The P-Bridge reestablishes connectivity so that fragments can be associated with their originating request, but it does not preserve call-graph topology or sibling ordering.

Breadcrumbs: The P-Bridge propagates only AMQ filters containing ancestor fingerprints (Table 6.3), the same vertical breadcrumbs used by the CG-Bridge, but without the fan-out arrays. Since we only need to establish connectivity (not reconstruct topology), AMQ filters alone are sufficient.

Reconstruction: When spans are lost, the P-Bridge identifies the nearest surviving upstream span by querying the AMQ filter for known ancestors at shallower depths. Synthetic spans are inserted to bridge the gap between the disconnected fragment and its nearest ancestor. Unlike the CG-Bridge, no fan-out merging is performed. If multiple children of a lost parent survive, they will each connect to their own synthetic parent rather than a shared one.

Overhead: The P-Bridge has the lowest overhead of the three bridges. Breadcrumb size depends only on call depth. Reconstruction requires only AMQ filter queries. This makes the P-Bridge suitable for high-volume tracing where full topology reconstruction is unnecessary.

6.3.5 Managing Overhead

Breadcrumbs accumulate as requests traverse deeper call paths, and reconstruction requires that at least some spans along each path are successfully collected, specifically all leaf nodes. This section discusses two mechanisms for managing these costs.

Checkpointing: Rather than storing breadcrumbs in every span, bridges support *checkpointing*: only designated checkpoint spans persist breadcrumb data, and the breadcrumb state resets after each checkpoint. This bounds breadcrumb size and reduces collection overhead.

Checkpoint spans serve as anchors for reconstruction. The algorithms presented in this section assume all nodes are checkpointed (i.e. contain breadcrumbs), but the approach generalizes to intermediate checkpoints. Reconstruction simply operates on subgraphs between consecutive checkpoints.

Checkpoint distance (the number of spans between checkpoints) can be configured statically or dynamically. A static policy might checkpoint every k spans on a call path. A dynamic policy can adapt to system load: if the span at depth k is under high load (e.g., its export queue exceeds a threshold), checkpointing shifts to the next downstream span. Checkpoint distance can also be slightly randomized to avoid synchronized bursts of checkpoint traffic across services.

Leaf collection: Bridges can only reconstruct paths to spans that were successfully collected. If a leaf span is lost, no downstream breadcrumbs exist to enable reconstruction of this node. This is problematic because leaf services (e.g., databases, caches) often experience the highest load and are most likely to drop spans.

One mitigation is to prioritize leaf span collection by stripping non-essential attributes from leaf spans, retaining only the fingerprint and breadcrumb data. This reduces export size and increases the likelihood of successful collection under load. Alternatively, services can be configured to always checkpoint before calling known high-load leaves, ensuring that even if the leaf span is lost, the path up to the checkpoint is recoverable.

6.4 Supporting bridges in OTEL

This section describes how bridges can be integrated into existing tracing infrastructure. We present Manchac, a prototype implementation built on OpenTelemetry (OTEL) that demonstrates the feasibility of bridge-augmented tracing with minimal modifications to application code.

Figure 6.7 illustrates the architecture. Manchac consists of two planes: a *data plane* that augments spans with breadcrumbs during request execution, and an *analysis plane* that detects data loss and reconstructs traces offline. The data plane integrates with OTEL's SDK and collector infrastructure; the analysis plane operates on traces stored in a backend database.

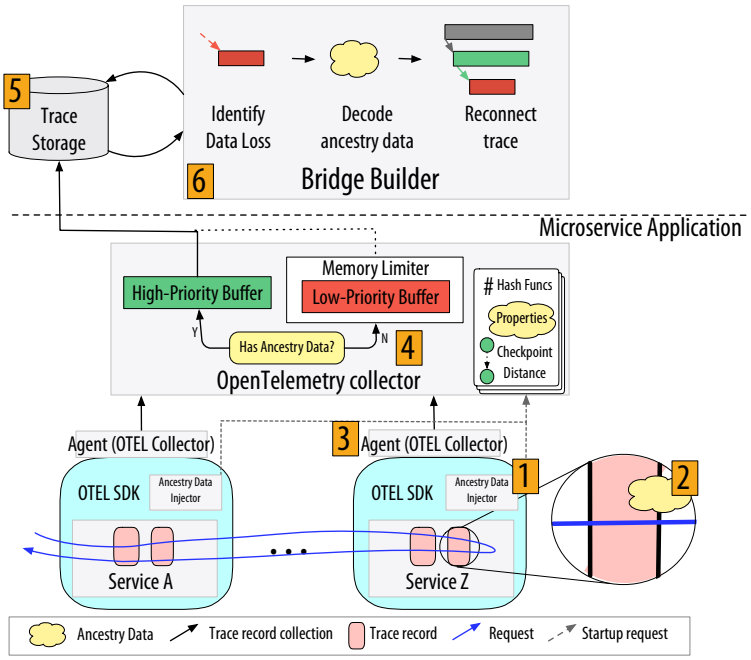


Figure 6.7: **Holes design diagram.** Separates tracing data as low and high priority with ancestry information to mitigate the affects of data loss

6.4.1 Manchac Overview

The data plane comprises three components, corresponding to the numbered steps in Figure 6.7:

SDK-level processor (Steps 1–2): Each service is instrumented with an OTEL SDK augmented with a custom span processor. The SDK-level processor is initialized with the checkpoint distance and the set of hash functions to be used for bloom filters and hash arrays. On span creation, the processor computes breadcrumb data (bloom filters, fan-out arrays, or ordinals depending on bridge type) and attaches it to the span's baggage. When the service issues downstream calls, the SDK propagates this baggage in-band with the request context, where downstream processors extend the breadcrumbs with their own span information.

Collector agent (Steps 3–4): An OTEL collector deployed as an agent on each node receives spans from local services. A custom processor partitions incoming spans into high-priority (checkpoint spans containing breadcrumbs) and low-priority (non-checkpoint spans) queues. Under memory pressure, low-priority spans are dropped to ensure checkpoint spans survive.

Trace Storage Backend (Steps 5–6): Spans exported from agents pass through a second collector instance before reaching the trace storage backend. This collector applies the same priority filtering, providing a final safeguard against checkpoint span loss during load spikes. The analysis plane monitors stored traces; when it detects incomplete traces (disconnected fragments), it invokes the appropriate reconstruction algorithm and writes the repaired traces back to storage.

6.4.2 Checkpointing

Checkpointing bounds breadcrumb size and collection overhead by designating specific spans as checkpoints. Only checkpoint spans persist breadcrumb data; after a checkpoint, the breadcrumb state resets.

Checkpoint distance: The *checkpoint distance* D specifies the maximum number of spans between checkpoints on any call path. A span at depth kD (for integer k) becomes a checkpoint, stores its accumulated breadcrumbs as a span attribute, and resets the breadcrumb state for downstream spans. All root and leaf spans are automatically checkpointed to ensure complete coverage of each call path.

Priority levels: Checkpoint spans are marked high-priority and routed through a dedicated collection pipeline that resists dropping under load. Non-checkpoint spans are low-priority and may be dropped when collectors exceed memory thresholds. This two-tier approach ensures that the minimal data required for reconstruction (the checkpoint spans) survives even during severe data loss events.

6.4.3 OTEL Integration

We implement Manchac by adding custom processors to OTEL's SDK and collector. SDK processors generate and propagate breadcrumbs and collector processors implement priority-based filtering. Both are drop-in components requiring no modifications to application code beyond standard OTEL instrumentation.

SDK processor: The SDK processor is initialized with the bridge type, checkpoint distance D, and hash functions (for AMQ filters). On span start, it:

1. Extracts existing breadcrumb data and call depth from incoming baggage
2. If depth mod $D = 0$, marks the span as a checkpoint and serializes breadcrumbs to a span attribute. Resets the breadcrumb to empty.
3. Adds the current span's fingerprint to the breadcrumb structure
4. Propagates updated breadcrumbs and depth in outgoing baggage

Configuration (bridge type, D, hash functions) is obtained from a central configuration service at startup, ensuring consistency across all services. Bloom filters are used as the AMQ filter, and their size is set to a target false positive rate of 0.1% for D elements.

Collector processor: The collector processor maintains two queues partitioned by span priority. A configurable memory threshold (e.g., 90%) triggers dropping of low-priority spans. High-priority spans are never dropped by the processor, though they may still be lost due to network failures or backend unavailability. The same processor runs in both the per-node agent and the backend collector for defense in depth.

6.5 Preliminary Evaluation

This section presents an initial evaluation of bridges. We evaluate two aspects: (1) the runtime overhead of Manchac when deployed on a microservice testbed, and (2) the data overhead of breadcrumbs across different trace topologies, measured by annotating production traces from Uber and synthetically generated traces to isolate topological characteristics. Full evaluation is ongoing work.

6.5.1 Experimental Setup

Testbed deployment: We deployed DeathStarBench's Social Network application using the Blueprint microservice compiler framework on Kubernetes. We deployed on 4 CloudLab nodes, each with 56 cores and 256GB of memory, with one control plane node and three worker nodes.

Instrumentation: Services are instrumented with OpenTelemetry, extended with Manchac's custom span processors to compute and propagate breadcrumbs. OpenTelemetry collectors run as agents on each node, receiving spans and forwarding them to centralized storage.

Trace datasets: We use Uber's open-source traces and synthetic traces. Uber's traces provide realistic production topologies; synthetic traces isolate the impact of specific topology parameters (depth, width) on breadcrumb overhead.

6.5.2 In-band Latency Overhead

We measure the end-to-end latency impact of computing and propagating breadcrumbs during request execution.

Methodology: We load test the Social Network's ComposePost endpoint using the open-loop *wrk2* load generator. ComposePost exercises the most complex workflow in the application, with call depths up to 7 and fan-out degrees up to 7, providing a stress test for breadcrumb overhead. We vary request rates from 50 to 800 requests per second and compare against two baselines: no tracing and vanilla OpenTelemetry tracing (no bridges). We evaluate all bridge types (Structural, Call-Graph, Path) with checkpoint distance **MISSING_VAL 11**.

Results: Figure 6.8 shows mean response times across request rates. Up to the saturation point (750 req/s), breadcrumb overhead is negligible across all bridge types and checkpoint distances. Response times diverge only under saturation, when queuing effects amplify small per-request differences.

The minimal overhead is expected: breadcrumb computation involves lightweight operations (hashing, array appends) that complete in microseconds, while application work and span serialization dominate request latency. The additional bytes added to trace context fit within existing network packets alongside application payloads, avoiding extra round trips.

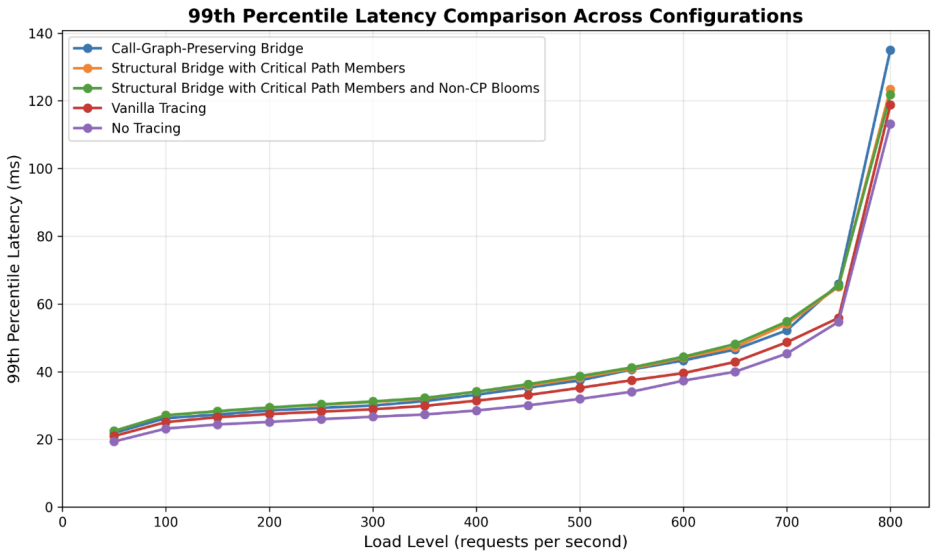


Figure 6.8: **Latency overhead of adding breadcrumbs to spans.** for various bridge types, including no tracing and vanilla tracing (with no bridges)

6.5.3 Breadcrumb Data Size

We evaluate how breadcrumb size varies with checkpoint distance and trace topology.

Checkpoint distance: Figure 6.9 shows average breadcrumb size per span for Uber's traces across checkpoint distances 2–6. As expected, larger checkpoint distances reduce the number of checkpointed spans but increase breadcrumb size at each checkpoint (longer ancestry paths to encode). The Exact Bridge incurs the highest overhead due to ordering metadata; the Path Bridge is most compact.

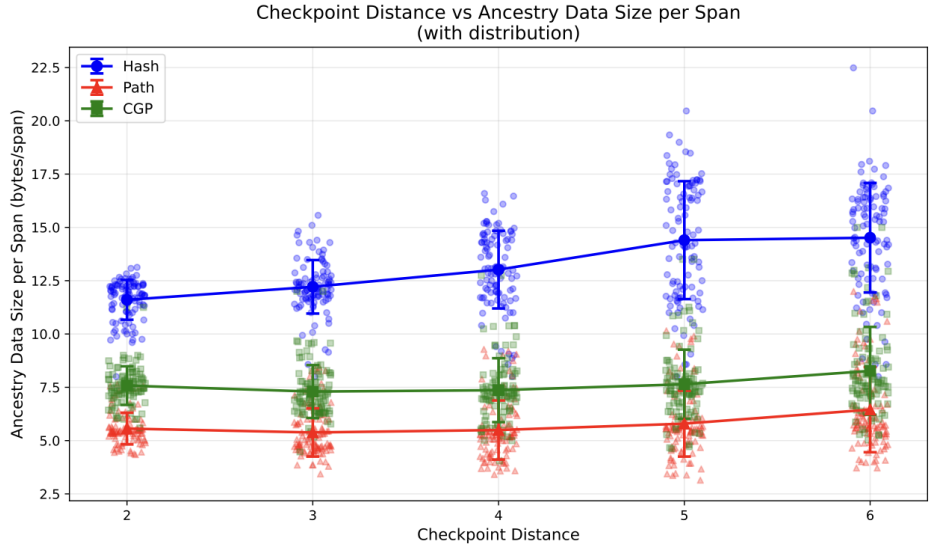
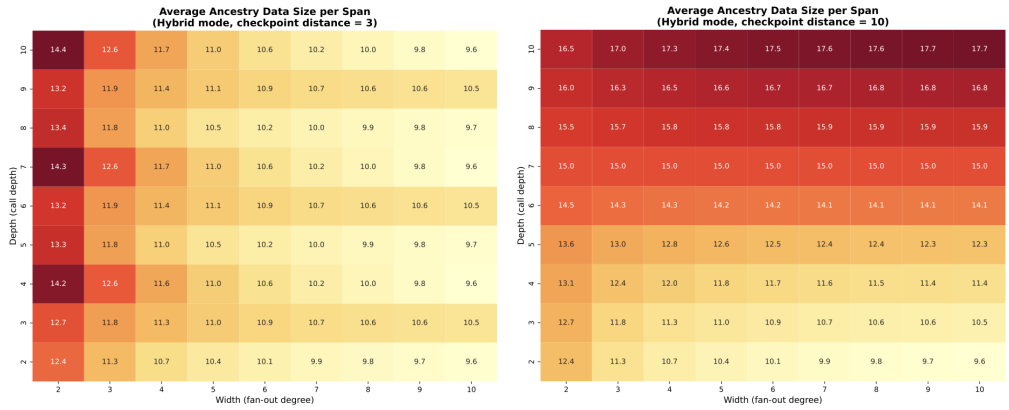


Figure 6.9: **Checkpoint distance vs Payload.** Average ancestry data (per span) for various bridge types.



(a) checkpoint distance 3 (b) checkpoint distance 10

Figure 6.10: **Breadcrumb size for various depth and width traces.** Plots amortized breadcrumb size per trace for various topologies.

Trace topology: Figure 6.10 shows breadcrumb size for synthetic traces varying in depth and width. We generate complete w-ary trees of depth d and measure average breadcrumb size per span.

With checkpoint distance 3 (Figure 6.10a), breadcrumb sizes remain small (max 14.4 bytes) because paths between checkpoints are short. Average size decreases with width because the ratio of leaf nodes (which carry breadcrumbs) to internal nodes grows, and leaf breadcrumbs encode only their path to the nearest checkpoint.

With checkpoint distance 10 (Figure 6.10b), only root and leaf nodes are checkpointed. Breadcrumb size increases with depth as paths grow longer. At depth 7, the trend reverses: as width increases, the fraction of nodes that are checkpointed approaches 1, and each carries substantial ancestry data (168 bytes).

Takeaway: Checkpoint distance provides a tunable tradeoff between breadcrumb frequency and size. For typical production traces (shallow, moderate width), checkpoint distances of 3–5 keep per-span overhead under 20 bytes while providing sufficient redundancy for reconstruction.

6.6 Conclusion

Data loss in distributed tracing is inevitable: infrastructure sheds non-critical work during overloads, sampling drops records to manage costs, and failures disrupt collection pipelines. Yet existing tracing systems provide no mechanism to recover connectivity when records are lost.

This chapter introduced bridges, a new primitive that preserves trace characteristics across holes by propagating lightweight breadcrumbs with requests. We presented three bridge types offering different fidelity-overhead tradeoffs: Structural Bridges preserve topology and sibling ordering for critical-path analysis, Call-Graph Bridges preserve topology with reduced overhead, and Path Bridges preserve only connectivity for minimal cost. Our preliminary

evaluation shows that breadcrumb overhead is negligible in terms of latency and modest in terms of data size, scaling predictably with checkpoint distance and trace topology.

Bridges represent a shift in how we think about trace reliability—from assuming complete data to designing for graceful degradation.

Chapter 7

Conclusion

7.1 Summary

This dissertation argues that understanding microservices requires moving beyond idealized models to confront the messy realities of production systems. Across qualitative studies and large-scale industrial analyses, the work presented here demonstrates that microservice architectures are far more dynamic, heterogeneous, and error-prone than commonly assumed and that observability infrastructures must be designed with these realities in mind. Rather than treating data loss as an edge case or assuming testbeds faithfully represent production, this research embraces the complexity and imperfection inherent to large-scale distributed systems. Looking ahead, this perspective opens opportunities for building more robust observability tools, more representative experimental platforms, and standardized methodologies for comparing microservice architectures across organizations.

7.2 Future Work

7.2.1 Extending Bridges

The bridges work presented in this dissertation is ongoing. Beyond completing the current implementation and evaluation, several directions emerge.

Inferring missing span metadata: Bridges recover trace topology but not the metadata including service names and endpoints associated with lost spans. Graph neural networks trained on complete traces could learn to predict likely service identities from topological context. Even partial inference would be valuable: knowing that a missing span is "probably a database call" could narrow debugging scope considerably.

Adapting trace-consuming tools: Anomaly detectors, auto-scalers, and critical-path analyzers assume complete traces. These tools must be adapted to be aware of data loss in traces and extended to handle reconstructed traces at varying levels of completeness. First, systems have various approaches to handling data loss in traces. Most commonly, the entire trace is omitted when it invalidates accepted trace properties (e.g. fully connected). There must be a principled way to acknowledge lossy trace data without discarding it completely. Second, tools should be extended to handle traces with bridges. Traces that lose data likely happen when the system is under high load or experiencing failures, making them the most beneficial for tools and developers. Tools should also

adapt to traces at varying granularity levels. For example, aggregation tools should be able to handle connected traces that may be missing nodes along bridges.

Bridge-aware sampling: Today, sampling is made an a full request granularity. Either the entire request is traced or it is not traced at all. Bridges decouple trace connectivity from completeness, enabling a new within trace sampling approach: intentionally drop low-value spans while preserving the trace skeleton via breadcrumbs. Operators could define verbosity levels like full fidelity for debugging, skeleton-only for end-to-end monitoring with bridges ensuring connectivity regardless. The challenge is identifying which spans are safe to drop for a given use case.

7.2.2 Improving Trace Data Quality

Detecting context-propagation errors: Capturing high-fidelity traces that represent their workflow relies on correct context propagation. Additionally, context propagation errors can propagate downstream, making it difficult to identify the offending service. Tools, similar to lint, are needed that can detect whether services are propagating context correctly. These tools should be used prior to deploying new services or new versions and could be integrated into CI/CD pipelines.

Redundancy in trace data models: CASPER's reconstruction works because Alibaba's model encodes redundant information. OpenTelemetry stores only a parent pointer in each span. When that parent is lost, connectivity breaks. Bridges add redundancy via propagated breadcrumbs, but other designs exist: span IDs could encode ancestry rather than being random 64 bit strings, or services could periodically checkpoint trace state. Each approach trades off overhead, reconstruction fidelity, and implementation complexity differently.

7.2.3 Microservice Experimentation

Testbeds that evolve: Our findings reveal that production microservices are characterized by constant change: services are deployed and deprecated daily, communication patterns shift, and workflows vary across executions. Yet existing testbeds are static snapshots. A more faithful experimental platform would incorporate evolution as a first-class property. Evolving testbeds would be useful to evaluate systems that train models of system characteristics to make decisions. The evolvability aspect can be used to evaluate how often models need to be retrained in order to keep up with the system changing. This raises design questions: How should evolution be parameterized? Should testbeds replay recorded evolution traces or generate synthetic change patterns? And how do we validate that synthetic evolution captures the properties that matter for the research questions at hand?

Generators as complements to testbeds: Full microservice deployments are infeasible outside industrial settings, making topology and workflow generators essential for research. But existing generators may encode assumptions specific to particular organizations or topologies. Research is needed to identify which architectural

dimensions are best explored via generators versus deployed testbeds, and how to calibrate generators against diverse production characteristics. Our finding that microservice topologies do not follow simple power-law relationships suggests that principled generative models perhaps learned from anonymized production data may be necessary.

7.2.4 Standardizing Measurement

Common vocabulary for comparison: Comparing Meta's architecture to prior studies proved difficult since definitions of "service" vary, complexity metrics differ, sampling methodologies are often unknown. Developing common definitions and measurement protocols would enable meaningful cross-organization comparisons.

Naming services and observability data: At Meta, we found that teams named services in unconventional ways to work around infrastructure limitations. Not all deployable units fit the standard service abstraction, so naming became a mechanism to coerce desired behavior. This ad-hoc approach leads to inconsistent naming quality, particularly for services satisfying many business use cases. Research into naming schemas that better accommodate diverse deployment patterns, and tools that surface meaningful names from observed behavior, would help both operators and automated tooling.

Bibliography

[1] Martin Fowler. Microservices: a definition of this new architectural term. `https://martinfowler.com/articles/microservices.html`.

[2] 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 *SoCC'21: Proceedings of the 12th Symposium on Cloud Computing*, 2021.

[3] Darby Huye, Yuri Shkuro, and Raja R. Sambasivan. Lifting the veil on Meta's microservice architecture: Aanalyses and topology and request workflows. In *ATC'23: Proceedings of the 2023 USENIX Annual Technical Conference*, 2023.

[4] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *ASPLOS'19: Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019.

[5] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chenjie Xu, Chao Ji, and Wenyun Zhao. Benchmarking microservice systems for software engineering research. In *ICSE Companion '14: Companion Proceedings of the 36th International Conference on Software Engineering*, 2018.

[6] Jóakim 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 *Proceedings of the 26th IEEE International Symposium on the Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, MASCOTS '18, September 2018.

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

[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*, SoCC '22, 2022.

[9] Chengzhi Lu, Huanle Xu, Kejiang Ye, Guoyao Xu, Liping Zhang, Guodong Yang, and Chengzhong Xu. Understanding and optimizing workloads for unified resource management in large cloud platforms. In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, 2023.

[10] Darby Huye, Lan Liu, and Raja R. Sambasivan. Systemizing and mitigating topological inconsistencies in alibaba's microservice call-graph datasets. In *ACM/SPEC International Conference on Performance Engineering*. ACM/SPEC, May 2024.

[11] Mario Villamizar, Oscar Garcés, Lina Ochoa, Harold Castro, Lorena Salamanca, Mauricio Verano, Rubby Casallas, Santiago Gil, Carlos Valencia, Angee Zambrano, and Mery Lang. Cost comparison of running web applications in the cloud using monolithic, microservice, and AWS lambda architectures. *Service Oriented Computing and Applications*, 11(2):233–247, April 2017.

[12] Mario Villamizar, Oscar Garcés, Harold Castro, Mauricio Verano, Lorena Salamanca, Rubby Casallas, and Santiago Gil. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In *2015 10th Computing Colombian Conference (10CCC)*, pages 583–590, 2015.

[13] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7), July 1978.

[14] OpenTelemetry. `https://opentelemetry.io/`.

[15] Microservices at Netflix Scale - First Principles, Tradeoffs & Lessons Learned. https://gotocon.com/amsterdam-2016/presentation/Microservices%20at%20Netflix%20Scale%20-%20First%20Principles,%20Tradeoffs%20&%20Lessons%20Learned.

[16] Olaf Zimmermann. Microservices tenets. *Computer Science - Research and Development*, 32(3):301–310, Jul 2017.

[17] Lalit Kale Gaurav Aroraa and Kanwar Manish. *Building Microservices with .NET Core*. Packtpub, USA, 2017.

[18] The Great Migration: from Monolith to Service-Oriented. https://www.infoq.com/presentations/airbnb-soa-migration/.

[19] Davide Taibi, Valentina Lenarduzzi, and Claus Pahl. Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation. *IEEE Cloud Computing*, 4(5):22–32, 2017.

[20] Paolo Di Francesco, Patricia Lago, and Ivano Malavolta. Migrating towards microservice architectures: An industrial survey. In *2018 IEEE International Conference on Software Architecture (ICSA)*, pages 29–2909, 2018.

[21] Nuha Alshuqayran, Nour Ali, and Roger Evans. Towards micro service architecture recovery: An empirical study. In *2018 IEEE International Conference on Software Architecture (ICSA)*, pages 47–4709, 2018.

[22] Antonio Bucchiarone, Nicola Dragoni, Schahram Dustdar, Stephan T. Larsen, and Manuel Mazzara. From monolithic to microservices: An experience report from the banking domain. *IEEE Software*, 35(3):50–55, 2018.

[23] Javad Ghofrani and Daniel Lübke. Challenges of microservices architecture: A survey on the state of the practice. 05 2018.

[24] Hulya Vural, Murat Koyuncu, and Sinem Guney. A systematic literature review on microservices. In Osvaldo Gervasi, Beniamino Murgante, Sanjay Misra, Giuseppe Borruso, Carmelo M. Torre, Ana Maria A.C. Rocha, David Taniar, Bernady O. Apduhan, Elena Stankova, and Alfredo Cuzzocrea, editors, *Computational Science and Its Applications – ICCSA 2017*, pages 203–217, Cham, 2017. Springer International Publishing.

[25] Luiz Carvalho, Alessandro Garcia, Wesley K. G. Assunção, Rafael de Mello, and Maria Julia de Lima. Analysis of the criteria adopted in industry to extract microservices. In *2019 IEEE/ACM Joint 7th International Workshop on Conducting Empirical Studies in Industry (CESI) and 6th International Workshop on Software Engineering Research and Industrial Practice (SER IP)*, pages 22–29, 2019.

[26] Holger Knoche and Wilhelm Hasselbring. Drivers and barriers for microservice adoption - a survey among professionals in germany. 14:1–35, 01 2019.

[27] Shanshan Li, He Zhang, Zijia Jia, Zheng Li, Cheng Zhang, Jiaqi Li, Qiuya Gao, Jidong Ge, and Zhihao Shan. A dataflow-driven approach to identifying microservices from monolithic applications. *Journal of Systems and Software*, 157:110380, 2019.

[28] ServiceCutter: A Structured Way to Service Decomposition. https://servicecutter.github.io/.

[29] Luciano Baresi, Martin Garriga, and Alan De Renzis. Microservices identification through interface analysis. In Flavio De Paoli, Stefan Schulte, and Einar Broch Johnsen, editors, *Service-Oriented and Cloud Computing*, pages 19–33, Cham, 2017. Springer International Publishing.

[30] Yuyang Wei, Yijun Yu, Minxue Pan, and Tian Zhang. A feature table approach to decomposing monolithic applications into microservices. In *12th Asia-Pacific Symposium on Internetware*, Internetware'20, page 21–30, New York, NY, USA, 2020. Association for Computing Machinery.

[31] Markos Viggiato, Ricardo Terra, Henrique Rocha, Marco Tulio Valente, and Eduardo Figueiredo. Microservices in practice: A survey study. *CoRR*, abs/1808.04836, 2018.

[32] Jacopo Soldani, Damian Andrew Tamburri, and Willem-Jan Van Den Heuvel. The pains and gains of microservices: A systematic grey literature review. *Journal of Systems and Software*, 146:215–232, 2018.

[33] Alan Bandeira, Carlos Alberto Medeiros, Matheus Paixao, and Paulo Henrique Maia. We need to talk about microservices: An analysis from the discussions on stackoverflow. In *Proceedings of the 16th International Conference on Mining Software Repositories*, MSR '19, page 255–259. IEEE Press, 2019.

[34] Muhammad Waseem, Peng Liang, and Mojtaba Shahin. A systematic mapping study on microservices architecture in devops. *Journal of Systems and Software*, 170:110798, 2020.

[35] Nuha Alshuqayran, Nour Ali, and Roger Evans. A systematic mapping study in microservice architecture. In *2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA)*, pages 44–51. IEEE, 2016.

[36] 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*, 2018.

[37] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chenjie Xu, Chao Ji, and Wenyun Zhao. Benchmarking microservice systems for software engineering research. In Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman, editors, *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 323–324. ACM, 2018.

[38] Lianping Chen. Microservices: Architecting for continuous delivery and devops. 03 2018.

[39] Justus Bogner, Jonas Fritzsch, Stefan Wagner, and Alfred Zimmermann. Assuring the evolvability of microservices: Insights into industry practices and challenges. *CoRR*, abs/1906.05013, 2019.

[40] Davide Taibi, Valentina Lenarduzzi, and Claus Pahl. *Microservices Anti-patterns: A Taxonomy*, pages 111–128. Springer International Publishing, Cham, 2020.

[41] Davide Taibi and Valentina Lenarduzzi. On the definition of microservice bad smells. *IEEE software*, 35(3):56–62, 2018.

[42] H. Zhang, S. Li, Z. Jia, C. Zhong, and C. Zhang. Microservice architecture in reality: An industrial inquiry. In *2019 IEEE International Conference on Software Architecture (ICSA)*, pages 51–60, Los Alamitos, CA, USA, mar 2019. IEEE Computer Society.

[43] Yingying Wang, Harshavardhan Kadiyala, and Julia Rubin. Promises and challenges of microservices: an exploratory study. *Empirical Software Engineering*, 26(4):63, May 2021.

[44] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. 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*, ASPLOS '19, page 3–18, New York, NY, USA, 2019. Association for Computing Machinery.

[45] 2024.

[46] Istio Service Mesh. `https://istio.io/`.

[47] A. Sriraman and T. F. Wenisch. μsuite: A benchmark suite for microservices. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–12, 2018.

[48] Yanqi Zhang, Weizhe Hua, Zhuangzhuang Zhou, G. Edward Suh, and Christina Delimitrou. Sinan: Ml-based and qos-aware resource management for cloud microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, page 167–181, New York, NY, USA, 2021. Association for Computing Machinery.

[49] 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*, ASPLOS 2021, page 135–151, New York, NY, USA, 2021. Association for Computing Machinery.

[50] Zhipeng Jia and Emmett Witchel. Nightcore: Efficient and scalable serverless computing for latency-sensitive, interactive microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, page 152–166, New York, NY, USA, 2021. Association for Computing Machinery.

[51] Nikita Lazarev, Shaojie Xiang, Neil Adit, Zhiru Zhang, and Christina Delimitrou. Dagger: Efficient and fast rpcs in cloud microservices with near-memory reconfigurable nics. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, page 36–51, New York, NY, USA, 2021. Association for Computing Machinery.

[52] 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. USENIX Association, November 2020.

[53] Lexiang Huang and Timothy Zhu. tprof: Performance profiling via structural aggregation and automated analysis of distributed systems traces. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, page 76–91, New York, NY, USA, 2021. Association for Computing Machinery.

[54] Amirhossein Mirhosseini, Sameh Elnikety, and Thomas F. Wenisch. *Parslo: A Gradient Descent-Based Approach for Near-Optimal Partial SLO Allotment in Microservices*, page 442–457. Association for Computing Machinery, New York, NY, USA, 2021.

[55] Rolando Brondolin and Marco D. Santambrogio. A black-box monitoring approach to measure microservices runtime performance. *ACM Trans. Archit. Code Optim.*, 17(4), nov 2020.

[56] Compare gRPC services with HTTP APIs. `https://docs.microsoft.com/en-us/aspnet/core/grpc/comparison?view=aspnetcore-5.0`.

[57] Google's GRPC vs REST Blog. `https://cloud.google.com/blog/products/application-development/rest-vs-rpc-what-problems-are-you-trying-to-solve-with-your-apis`.

[58] gRPC vs. REST: How Does gRPC Compare with Traditional REST APIs? `https://blog.dreamfactory.com/grpc-vs-rest-how-does-grpc-compare-with-traditional-rest-apis/#:~:text=%E2%80%9CgRPC%20is%20roughly%207%20times,HTTP%2F2%20by%20gRPC.%E2%80%9D`.

[59] 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*, page 412–426. Association for Computing Machinery, New York, NY, USA, 2021.

[60] MicroSuite GitHub Repo. `https://github.com/wenischlab/MicroSuite/blob/master/install.py`.

[61] Istio BookInfo GitHub Repo. `https://github.com/istio/istio/blob/master/samples/bookinfo/src/build-services.sh`.

[62] TeaStore GitHub Repo. `https://github.com/DescartesResearch/TeaStore/tree/e189dff4d5cf3681a9b0b83f90b69c681dfd11da/services`.

[63] TrainTicket GitHub YAML. `https://github.com/FudanSELab/train-ticket/blob/350f62000e6658e0e543730580c599d8558253e7/docker-compose.yml`.

[64] DeathStarBench - Social Network - GitHub YAML. `https://github.com/delimitrou/DeathStarBench/blob/676a3b37811f580e39e50e17066af642ef895aa4/socialNetwork/docker-compose.yml`.

[65] DeathStarBench - Hotel Researvation - GitHub YAML. https://github.com/delimitrou/DeathStarBench/blob/676a3b37811f580e39e50e17066af642ef895aa4/hotelReservation/docker-compose.yml.

[66] DeathStarBench - Movie Recommendation - GitHub YAML. https://github.com/delimitrou/DeathStarBench/blob/676a3b37811f580e39e50e17066af642ef895aa4/mediaMicroservices/docker-compose.yml.

[67] Shang-Pin Ma, I-Hsiu Liu, Chun-Yu Chen, Jiun-Ting Lin, and Nien-Lin Hsueh. Version-based microservice analysis, monitoring, and visualization. In *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*, pages 165–172, 2019.

[68] How to design and version APIs for microservices (part 6). https://www.ibm.com/cloud/blog/rapidly-developing-applications-part-6-exposing-and-versioning-apis.

[69] Raja R. Sambasivan, Ilari Shafer, Jonathan Mace, Benjamin H. Sigelman, Rodrigo Fonseca, and Gregory R. Ganger. Principled workflow-centric tracing of distributed systems. In *ACM Symposium on Cloud Computing*, pages 401–414. ACM, October 2016.

[70] OpenTelemetry Website. https://opentelemetry.io/.

[71] WRK2 Workload Generator. https://github.com/giltene/wrk2.

[72] Aditya Vashistha, Edward Cutrell, and William Thies. Increasing the reach of snowball sampling: The impact of fixed versus lottery incentives. In *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work & Social Computing*, CSCW '15, page 1359–1363, New York, NY, USA, 2015. Association for Computing Machinery.

[73] Rodrigo Laigner, Yongluan Zhou, Marcos Antonio Vaz Salles, Yijian Liu, and Marcos Kalinowski. Data management in microservices: State of the practice, challenges, and research directions. *CoRR*, abs/2103.00170, 2021.

[74] William Viktorsson, Cristian Klein, and Johan Tordsson. Security-performance trade-offs of kubernetes container runtimes. In *2020 28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 1–4, 2020.

[75] Silvia Esparrachiari, Tanya Reilly, and Ashleigh Rentz. Tracking and controlling microservice dependencies: Dependency management is a crucial part of system and software design. *Queue*, 16(4):44–65, August 2018.

[76] Ben Treynor, Mike Dahlin, Vivek Rau, and Betsy Beyer. The calculus of service availability. *Commun. ACM*, 60(9):42–47, aug 2017.

[77] Hao Zhou, Ming Chen, Qian Lin, Yong Wang, Xiaobin She, Sifan Liu, Rui Gu, Beng Chin Ooi, and Junfeng Yang. Overload control for scaling wechat microservices. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, page 149–161, New York, NY, USA, 2018. Association for Computing Machinery.

[78] Universal Description, Discovery and Integration (UDDI) Registry. https://access.redhat.com/documentation/en-us/jboss_enterprise_soa_platform/5/html/esb_services_guide/universal_description_discovery_and_integration_uddi_registry.

[79] gRPC vs. REST: Performance Simplified. https://medium.com/@bimeshde/grpc-vs-rest-performance-simplified-fd35d01bbd4.

[80] gRPC vs REST — performance comparison. https://medium.com/analytics-vidhya/grpc-vs-rest-performance-comparison-1fe5fb14a01c.

[81] Akshitha Sriraman and Thomas F. Wenisch. µtune: Auto-tuned threading for OLDI microservices. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 177–194, Carlsbad, CA, October 2018. USENIX Association.

[82] Qingyang Wang, Chien-An Lai, Yasuhiko Kanemasa, Shungeng Zhang, and Calton Pu. A study of long-tail latency in n-tier systems: Rpc vs. asynchronous invocations. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 207–217, 2017.

[83] Comparing gRPC Performance. `https://www.nexthink.com/blog/comparing-grpc-performance/`.

[84] Irwin Kwan, Marcelo Cataldo, and Daniela Damian. Conway's law revisited: The evidence for a task-based perspective. *IEEE Software*, 29(1):90–93, 2012.

[85] Nachiappan Nagappan, Brendan Murphy, and Victor Basili. The influence of organizational structure on software quality: An empirical case study. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, page 521–530, New York, NY, USA, 2008. Association for Computing Machinery.

[86] Gopal Kakivaya, Lu Xun, Richard Hasha, Shegufta Bakht Ahsan, Todd Pfleiger, Rishi Sinha, Anurag Gupta, Mihail Tarta, Mark Fussell, Vipul Modi, Mansoor Mohsin, Ray Kong, Anmol Ahuja, Oana Platon, Alex Wun, Matthew Snider, Chacko Daniel, Dan Mastrian, Yang Li, Aprameya Rao, Vaishnav Kidambi, Randy Wang, Abhishek Ram, Sumukh Shivaprakash, Rajeet Nair, Alan Warwick, Bharat S. Narasimman, Meng Lin, Jeffrey Chen, Abhay Balkrishna Mhatre, Preetha Subbarayalu, Mert Coskun, and Indranil Gupta. Service fabric: A distributed platform for building microservices in the cloud. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[87] Akshitha Sriraman and Abhishek Dhanotia. Accelerometer: Understanding acceleration opportunities for data center overheads at hyperscale. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 733–750, New York, NY, USA, 2020. Association for Computing Machinery.

[88] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. Microservices architecture enables devops: Migration to a cloud-native architecture. *IEEE Software*, 33(3):42–52, 2016.

[89] Sara Hassan, Rami Bahsoon, and Rick Kazman. Microservice transition and its granularity problem: A systematic mapping study. 50(9):1651–1681, June 2020.

[90] Supreeth Shastri, Vinay Banakar, Melissa Wasserman, Arun Kumar, and Vijay Chidambaram. Understanding and benchmarking the impact of GDPR on database systems. *PVLDB*, 13(7):1064–1077, 2020.

[91] Ghulam Murtaza, Amir R Ilkhechi, and Saim Salman. Impact of gdpr on service meshes.

[92] John Jenkins, Galen Shipman, Jamaludin Mohd-Yusof, Kipton Barros, Philip Carns, and Robert Ross. A case study in computational caching microservices for hpc. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1309–1316, 2017.

[93] 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*, ASPLOS '19, page 19–33, New York, NY, USA, 2019. Association for Computing Machinery.

[94] Twitter Diffy GitHub. `https://github.com/twitter-archive/diffy`.

[95] Anelis Pereira-Vale, Eduardo B. Fernandez, Raúl Monge, Hernán Astudillo, and Gastón Márquez. Security in microservice-based systems: A multivocal literature review. *Computers & Security*, 103:102200, 2021.

[96] Yingying Wen, Guanjie Cheng, Shuiguang Deng, and Jianwei Yin. Characterizing and synthesizing the workflow structure of microservices in ByteDance cloud. *Journal of Software: Evolution and Process*, 34(8):1–18, 2022.

[97] Sam Newman. *Building microservices: designing fine-grained systems*. O'Reilly Media, Inc., 2nd edition, 2021.

[98] Yu Gan, Mingyu Liang, Sundar Dev, David Lo, and Christina Delimitrou. Sage: practical and scalable ML-driven performance debugging in microservices. In *ASPLOS'21: Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 135–151, 2021.

[99] 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 *OSDI'20: Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation*, pages 805–825, 2020.

[100] Yanqi Zhang, Weizhe Hua, Zhuangzhuang Zhou, G Edward Suh, and Christina Delimitrou. Sinan: ML-based and qos-aware resource management for cloud microservices. In *ASPLOS'21: Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 167–181, 2021.

[101] Lexiang Huang and Timothy Zhu. tprof: Performance profiling via structural aggregation and automated analysis of distributed systems traces. In *SoCC'21: Proceedings of the 12th Symposium on Cloud Computing*, 2021.

[102] Zhizhou Zhang, Murali Krishna Ramanathan, Prithvi Raj, Abhishek Parwal, Timothy Sherwood, and Milind Chabbi. CRISP: Critical path analysis of Large-Scale microservice architectures. In *ATC'22: Proceedings of the 2022 USENIX Annual Technical Conference*, 2022.

[103] Harshit Saokar, Soteris Demetriou, Nick Magerko, Max Kontorovich, Josh Kirstein, Margot Leibold, Dimitrios Skarlatos, Hitesh Khandelwal, and Chunqiang Tang. ServiceRouter: a Scalable and Minimal Cost Service Mesh. In *OSDI'23: Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation*, 2023.

[104] Mark Slee, Aditya Agarwal, and Marc Kwiatkowski. Thrift: Scalable cross-language services implementation. *Facebook white paper*, 2007.

[105] Jonathan Kaldor, Jonathan Mace, Michał Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, Vinod Venkataraman, Kaushik Veeraraghavan, and Yee Jiun Song. Canopy: An end-to-end performance tracing and analysis system. In *SOSP '17: Proceedings of the 26th Symposium on Operating Systems Principles*, 2017.

[106] Raja R. Sambasivan, Ilari Shafer, Jonathan Mace, Benjamin H Sigelman, Rodrigo Fonseca, and Gregory R. Ganger. Principled workflow-centric tracing of distributed systems. In *SoCC '16: Proceedings of the Seventh Symposium on Cloud Computing*, 2016.

[107] Rodrigo Fonseca, Michael J. Freedman, and George Porter. Experiences with tracing causality in networked services. In *INM/WREN '10: Proceedings of the 2010 Internet Network Management Workshop/Workshop on Research on Enterprise Monitoring*, 2010.

[108] Raja R. Sambasivan, Alice X. Zheng, Michael De Rosa, Elie Krevat, Spencer Whitman, Michael Stroucken, William Wang, Lianghong Xu, and Gregory R. Ganger. Diagnosing performance changes by comparing request flows. In *NSDI'11: Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, 2011.

[109] Patrick Reynolds, Charles Killian, Janet L. Wiener, Jeffrey C. Mogul, Mehul Shah, and Amin Vahdat. Pip: detecting the unexpected in distributed systems. In *NSDI '06: Proceedings of the 3rd USENIX Symposium on Networked Systems Design and Implementation*, 2006.

[110] Benjamin H. Sigelman, Luiz A. Barroso, Michael Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical Report dapper-2010-1, Google, April 2010.

[111] Yuri Shkuro. Jaeger: Evolving distributed tracing at Uber Engineering. `https://www.uber.com/blog/distributed-tracing/`.

[112] Tracing in OpenTelemetry. `https://opentelemetry.io/docs/concepts/signals/traces/`.

[113] Yuan Mei, Luwei Cheng, Vanish Talwar, Michael Y Levin, Gabriela Jacques-Silva, Nikhil Simha, Anirban Banerjee, Brian Smith, Tim Williamson, Serhat Yilmaz, et al. Turbine: Facebook's service management platform for stream processing. In *ICDE'20: Proceedings of the 36th International Conference on Data Engineering*, pages 1591–1602. IEEE, 2020.

[114] Tyler Akidau, Slava Chernyak, and Reuven Lax. *Streaming systems: the what, where, when, and how of large-scale data processing*. O'Reilly Media, Inc., 2018.

[115] Yuri Shkuro. *Mastering Distributed Tracing: Analyzing performance in microservices and complex systems*. Packt Publishing, 2019.

[116] Vaastav Anand, Matheus Stolet, Thomas Davidson, Ivan Beschastnikh, Tamara Munzner, and Jonathan Mace. Aggregate-driven trace visualizations for performance debugging. *CoRR*, abs/2010.13681, 2020.

[117] Raja R. Sambasivan, Ilari Shafer, Michelle L Mazurek, and Gregory R. Ganger. Visualizing request-flow comparison to aid performance diagnosis in distributed systems. *IEEE Transactions on Visualization and Computer Graphics*, 19(12):2466–2475, 2013.

[118] Chenxi Zhang, Xin Peng, Chaofeng Sha, Ke Zhang, Zhenqing Fu, Xiya Wu, Qingwei Lin, and Dongmei Zhang. DeepTraLog: Trace-log combined microservice anomaly detection through graph-based deep learning. In *ICSE'22: Proceedings of the 44th International Conference on Software Engineering*, 2022.

[119] kubernetes. `https://kubernetes.io`.

[120] Introducing Domain-Oriented Microservice Architecture. `https://www.uber.com/blog/microservice-architecture/`.

[121] IMC: Internet Measurement Conference. `https://dl.acm.org/conference/imc`.

[122] 2021. `https://github.com/alibaba/clusterdata/tree/master/cluster-trace-microservices-v2021`.

[123] 2022. `https://github.com/alibaba/clusterdata/tree/master/cluster-trace-microservices-v2022`.

[124] Shutian Luo, Huanle Xu, Kejiang Ye, Guoyao Xu, Liping Zhang, Jian He, Guodong Yang, and Chengzhong Xu. Erms: Efficient resource management for shared microservices with sla guarantees. ASPLOS 2023, 2022.

[125] Xiangfeng Zhu, Guozhen She, Bowen Xue, Yu Zhang, Yongsu Zhang, Xuan Kelvin Zou, Xiongchun Duan, Peng He, Arvind Krishnamurthy, Matthew Lentz, Danyang Zhuo, and Ratul Mahajan. Dissecting service mesh overheads, 2022.

[126] 2021. `https://github.com/alibaba/clusterdata/issues/157`.

[127] 2022.

[128] 2023. `https://github.com/alibaba/clusterdata/issues/175`.

[129] 2022. `https://github.com/alibaba/clusterdata/issues/157`.

[130] 2024.

[131] Pedro Las-Casas, Giorgi Papakerashvili, Vaastav Anand, and Jonathan Mace. Sifter: Scalable sampling for distributed traces, without feature engineering. In *SoCC'19: Proceedings of the Ninth Symposium on Cloud Computing*, 2019.