Distribution Agreement

In presenting this thesis or dissertation as a partial fulfillment of the requirements for an advanced degree from Emory University, I hereby grant to Emory University and its agents the non-exclusive license to archive, make accessible, and display my thesis or dissertation in whole or in part in all forms of media, now or hereafter known, including display on the world wide web. I understand that I may select some access restrictions as part of the online submission of this thesis or dissertation. I retain all ownership rights to the copyright of the thesis or dissertation. I also retain the right to use in future works (such as articles or books) all or part of this thesis or dissertation.

Signature: DocuSigned by: Vishwanath Seshagiri 61CE06EDB2204D5...

Vishwanath Seshagiri
Name

4/21/2025 | 1:37 PM EDT
Date

The Whole Nine-nine Yard: Observability For The Observers

By
Vishwanath Seshagiri
Doctor of Philosophy

Computer Science and Informatics

Dr. Andreas Züfle
Advisor

Dr. Joon-Seok Kim
Committee Member

Dr. Ymir Vigfusson
Committee Member

Dr. Avani Wildani
Committee Member

Dr. Irene Zhang
Committee Member

Accepted:

Dr. Kimberly Jacob Arriola, PhD
Dean of the James T. Laney School of Graduate Studies

April 10, 2025

Date

The Whole Nine-nine Yard: Observability For The Observers

By

Vishwanath Seshagiri

Advisor: Dr. Andreas Züfle

An abstract of
A dissertation submitted to the Faculty of the
James T. Laney School of Graduate Studies of Emory University
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy
in Computer Science and Informatics
2025

Abstract

The Whole Nine-nine Yard: Observability For The Observers
By Vishwanath Seshagiri

The rapid adoption of microservice architectures in modern cloud computing has exposed critical limitations in traditional performance measurement methodologies and observability tools. This work demonstrates how the separation of development and operational responsibilities in modern enterprises renders conventional observability tools inadequate to modern software engineering workflows. We present a three-pronged approach to bridge this gap: First, through systematic characterization of industrial microservice deployments revealing critical design choice disparities with academic testbeds. Second, via the development of NL2QL a novel natural language interface and curated dataset that enables precise log query generation through fine-tuned language models, achieving up to 75% improvement in accuracy. Finally, by developing Sauron, a semantic search engine leveraging vector embeddings and retrieval-augmented generation to overcome terminology inconsistencies in log analysis, demonstrating 46.7-116.7% improvement in search relevance metrics.

This work establishes that next-generation observability tools must account for both technical complexity and human factors in distributed systems. By combining domain-specific language model fine-tuning with semantic search architectures, we show how to democratize access to observability data to be used beyond traditional logging use cases. The thesis contributes practical frameworks for performance analysis in microservice environments and provides empirical evidence that closing the academia-industry divide requires tooling adaptations mirroring real-world organizational structures and developer workflows.

The Whole Nine-nine Yard: Observability For The Observers

By

Vishwanath Seshagiri

Advisor: Dr. Andreas Züfle

A dissertation submitted to the Faculty of the
Emory College of Arts and Sciences of Emory University
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy
in Computer Science and Informatics
2025

Acknowledgments

I extend my deepest gratitude to my parents and brother, whose unwavering support has enabled me to pursue my goals and passions. Without their relentless encouragement and belief in me, this dissertation would not have been possible.

As they say, it takes a village to raise a child – similarly, this dissertation represents the collective effort of the academic community at Emory that nurtured my research journey. First and foremost, I am profoundly grateful to my committee members: Andreas, Avani, Irene, Joon, Nosayba, and Ymir. They have been pillars of strength throughout this process, offering guidance not only on research matters but also on navigating the challenging landscape of academia. Their steadfast support persisted regardless of whether papers were accepted, rejected, or deadlines missed.

I am equally indebted to my colleagues in the Simbiosys Lab – Yazhuo, Pranav, Reza, Lei, Gary, and Siand the Spatial Computing Lab – Hossein, Lance, Ruochen, and Alex. These exceptional collaborators provided invaluable feedback on early drafts and created a space where ideas could flourish. This research would not have been possible without the broader support of the CS Department and LGS staff.

During my doctoral journey, I have been fortunate to learn from what I affectionately call a "Consortium of Researchers": Kostis, Amy, Simon, Pedro, Vaastav, Jose, Prashant and Vahab. Special recognition goes to Deepti, Amanda, and Micah, who have essentially become honorary labmates. What began as a collaboration on building a cache blossomed into a comprehensive support network where we discuss far more than just research. Their willingness to listen to my reflections on academia, job searching, and research challenges was instrumental in helping me navigate the inherent uncertainties of graduate school.

A PhD truly embodies "the best of times and the worst of times," with constant oscillation between the two extremes. Throughout these fluctuations, I was blessed with an extraordinary group of friends at Emory: Ayush, Ylli, Mumi, Julia, Leo,

Vincent, Marcelo, Kelvin, Nat, Sean, Anton, Katie, Emma, Elle, Abbey, and Ishleen. Their presence was a constant, regardless of circumstances. Sometimes when faced with disappointment, all one needs is someone to lighten the mood with good-natured humor. From singing off-key renditions of Billy Joel's "Piano Man" to experimenting with new recipes, they have shared in all of life's moments with me.

Beyond Emory, I've connected with remarkable individuals – some adventurous enough to join me on various, occasionally precarious hikes: Mark, Tim, Aaron, Hayley, Francisco, Marianne, Jenny, Ashwath, Neo, Spencer, Cayden, Horizon, and Patrick. Thank you for filling my summers with joy and adventure!

Finally, my heartfelt thanks to Shruthi. From our joint move to the US to navigating our careers and the challenges of a long-distance relationship, you have been my constant companion in figuring things out. You are the bestest.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction

Computer Science, particularly the field of Systems, is fundamentally driven by optimizationa discipline where extracting even a millisecond of performance improvement can translate into billions of dollars in profit or cost savings for organizations[136]. This relentless pursuit of efficiency has historically been motivated by the imperative to maximize shareholder value, compelling researchers and practitioners to develop increasingly sophisticated tools and methodologies for measuring performance and identifying optimization opportunities[95, 96, 58, 56].

Performance measurement represents a foundational scientific endeavor with deep historical roots. From ancient civilizations developing the Royal Cubit[145] based on a pharaoh's hand dimensions to the modern establishment of standardized SI units, humanity has consistently sought to identify appropriate metrics and methodologies to quantify and advance scientific understanding. This quest for measurement precision has evolved dramatically in recent decades as computing paradigms have shifted from personal computers to distributed cloud environments – colloquially referred to as "someone else's computer" – introducing unprecedented complexity to performance measurement challenges[53]. The transition to cloud computing has catalyzed the widespread adoption of microservice architectures, wherein monolithic business applications are decomposed into smaller, modular services managed by discrete teams[15, 22, 13]. While this architectural approach offers significant advantages in terms of scalability, fault isolation, and development agility, it introduces substantial challenges for performance measurement and optimization. The distributed nature of these systems, coupled with their inherent complexity, has fundamentally transformed how researchers and practitioners approach performance analysis.

This architectural evolution has created a pronounced disconnect between academic research environments and industrial practice. While researchers have developed numerous benchmarks and testbeds to serve as proxies for real-world systems, these

academic artifacts frequently fail to capture the nuanced design choices, operational constraints, and organizational dynamics that characterize industry-scale deployments. Academic testbeds typically embody simplified assumptions about service interactions, dependency structures, and deployment patterns that may not align with the heterogeneity observed in production environments. The divergence between academic benchmarks and industrial reality is further exacerbated by the proprietary nature of many commercial microservice implementations. Organizations often develop highly customized architectures tailored to their specific business requirements, technical constraints, and organizational structures. These implementations frequently incorporate proprietary technologies, custom optimization strategies, and organization-specific design patterns that remain inaccessible to the broader research community. This information asymmetry limits the ability of researchers to develop truly representative benchmarks and evaluation frameworks.

Moreover, the observability challenges inherent in microservice architectures compound the difficulty of performance measurement. In traditional monolithic applications, performance bottlenecks could often be identified through relatively straightforward profiling and monitoring techniques. In contrast, microservice environments require sophisticated distributed tracing, log aggregation, and metrics collection systems to provide even basic visibility into system behavior. The organizational dynamics surrounding microservice development and operation further complicate performance optimization efforts. Unlike monolithic applications, which might be developed and maintained by a single team[135], microservice architectures typically involve multiple teams with distinct responsibilities, priorities, and technical approaches[65]. This organizational distribution can lead to inconsistent implementation patterns, varying quality standards, and communication challenges that impact overall system performance. Academic benchmarks rarely account for these human and organizational factors, despite their significant influence on real-world system behavior.

Despite these challenges, the imperative for effective performance measurement and optimization in microservice architectures remains paramount. As organizations continue to invest heavily in distributed systems, the economic impact of performance inefficiencies scales proportionally. Even modest improvements in service latency or resource utilization can yield substantial cost savings and competitive advantages when applied across large-scale deployments. This economic reality underscores the critical importance of developing more representative evaluation frameworks and measurement methodologies. Addressing this gap requires collaboration between academia and industry to create benchmarks and testbeds that more accurately reflect the complexity and diversity of real-world microservice deployments. Such collaboration would enable researchers to develop more relevant optimization techniques while providing practitioners with scientifically rigorous approaches to performance evaluation. By bridging this divide, the systems research community can drive meaningful advancements in the efficiency and reliability of modern distributed computing environments, ultimately delivering substantial value to organizations and also researchers.

1.1.1 Bridging the gap in Microservice testbeds

In the initial work of this thesis, we investigated the differences between academic testbeds and industrial deployments of microservices, identifying potential research questions to guide efforts aimed at improving the performance of testbeds. Industrial microservice architectures exhibit significant variability in characteristics such as size, communication methods, and dependency structures, making system comparisons challenging and often leading to confusion or misinterpretation. For instance, industrial systems frequently employ heterogeneous communication protocols (e.g., REST, gRPC) and mixed synchronous-asynchronous styles, while academic testbeds often adopt narrower configurations. Additionally, industrial systems may feature non-hierarchical dependency structures and even cyclic dependencies at the service or

endpoint levelphenomena rarely reflected in academic testbeds.

In contrast, academic testbeds used for microservices research tend to adopt a highly constrained set of design choices, often focusing on hierarchical topologies and limited service reuse. This lack of standardization in key design decisions when developing microservice architectures has created uncertainty regarding the applicability of testbed experiments to practical deployments and whether such extrapolation is appropriate. For example, while industrial systems may integrate shared services across multiple applications or employ diverse storage strategies (dedicated vs. shared), academic testbeds frequently lack such flexibility.

To address this issue, we conducted semi-structured interviews with industry professionals to evaluate the representativeness of existing testbed design choices. Notable findings included the presence of cycles in industrial deployments and a lack of clarity concerning hierarchical structures. Participants also emphasized challenges related to versioning support and testing practices, such as limited adoption of distributed tracing tools like Jaeger or Zipkin in academic environments. Based on insights from these interviews, we systematized the range of possible design choices and identified critical mismatches between the characteristics of industrial systems and those reflected in current testbeds. These findings will inform the development of future testbeds that are more representative of real-world microservice deployments. This work was published at Journal of Systems Research in 2022[116].

Our research revealed a critical disconnect between the developer tools ecosystem and the evolving microservice paradigm. The fundamental assumption underpinning traditional observability tools – that software developers are simultaneously responsible for deployment and production operations – no longer reflects organizational realities in modern enterprises. Despite industry efforts to adapt existing tooling, observability platforms remain largely anchored in outdated operational models that fail to address the complexities of contemporary software delivery pipelines. In today's microservice

environments, development and operations responsibilities are increasingly separated, with dedicated Site Reliability Engineering (SRE) teams often managing production deployments while development teams focus on feature implementation. This separation is further complicated by significant developer turnover within organizations, resulting in situations where those troubleshooting production issues may have limited familiarity with the original implementation details. The conventional observability tools, designed with monolithic architectures and stable team compositions in mind, struggle to bridge this knowledge gap effectively. Additionally, the scale and complexity of microservice architectures – characterized by distributed components, diverse communication patterns, and intricate dependency networks – demand a fundamental re-imagining of observability approaches. Current tools typically require deep expertise in platform specific query languages and intimate knowledge of application internals, creating substantial cognitive overhead for developers and operations personnel alike. This expertise barrier becomes particularly problematic in environments with high developer churn or organizational boundaries between development and operations teams.

These findings underscore the urgent need for next-generation observability tools specifically designed for the microservice eratools that can facilitate knowledge transfer across team boundaries, reduce reliance on specialized expertise, and adapt to the dynamic nature of modern software organizations. Such tools must prioritize accessibility, knowledge preservation, and cross-functional collaboration to effectively support the distributed nature of both microservice architectures and the teams that build and maintain them.

1.1.2 Chatting with Logs

Building on one of the findings from the prior analysis highlights the significant challenges developers face when analyzing observability data collected through diverse

tools. This insight forms the foundation for the second chapter of this thesis. Logging, an indispensable component of modern distributed systems, suffers from a lack of standardization in log query languages and formats, creating considerable obstacles for developers. Developers are often required to craft ad hoc queries using platform-specific log query languages, necessitating not only expertise in these languages but also a deep understanding of application-specific log details. Given the diversity of platforms and the vast volume of logs and applications, this expectation is frequently impractical.

While large language models (LLMs) offer promise in generating such queries, our findings reveal that existing LLMs struggle with this task due to limited exposure to domain-specific log-related knowledge. To address these challenges, we propose a novel natural language (NL) interface designed to mitigate inconsistencies and facilitate query generation. This interface allows developers to create queries in a target log query language by providing natural language inputs. Additionally, we introduce **NL2QL**, a manually curated dataset comprising real-world natural language questions paired with corresponding `logQL` queries, spanning three distinct log formats. NL2QL supports the development, training, and evaluation of NL-to-query systems.

Using NL2QL, we fine-tune and evaluate several state-of-the-art LLMs, demonstrating their improved ability to generate accurate `logQL` queries. Our experiments reveal up to 75% improvement in query-generation performance for fine-tuned models compared to non-fine-tuned counterparts. Furthermore, ablation studies assess the impact of additional training data and explore model transferability across different log formats. These results underscore the potential of fine-tuning LLMs for domain-specific tasks.

The NL2QL dataset represents a pioneering effort in creating a resource specifically tailored for NL-to-log-query systems. It includes 424 manually annotated pairs derived from Grafana dashboards, covering diverse applications such as OpenSSH, OpenStack,

and HDFS. This diversity ensures that the dataset captures various use cases and operational complexities encountered in real-world scenarios. Fine-tuning LLMs on this dataset significantly enhances their ability to generate syntactically valid and semantically accurate `LogQL` queries. Our evaluation framework employs metrics such as exact match accuracy and execution accuracy to assess query correctness. The fine-tuned models demonstrate substantial improvements in both syntactic precision and semantic relevance. Notably, GPT-4o achieves up to 80% accuracy in generating executable queries post-fine-tuning. These enhancements reduce syntax errors, improve label matching, and enhance temporal aggregation capabilities.

In addition to quantitative improvements, our study explores qualitative aspects such as cross-application transferability and the effect of varying training sample sizes. Results indicate that while fine-tuned models perform well within their training domains, generalization across applications remains challenging without additional diverse training data. In conclusion, this work demonstrates that leveraging LLMs for log query generation is not only feasible but also highly effective when supported by domain-specific datasets like NL2QL. This approach has the potential to democratize access to observability data by reducing reliance on specialized knowledge, thereby enhancing productivity and accessibility for developers. This work was submitted to VLDB 2025 as is now available on ArXiv[117].

While the NL2QL approach significantly improves developers' ability to generate log queries using natural language, it reveals certain limitations that necessitate a more comprehensive solution. Natural language querying, as implemented in the NL2QL system, offers developers a more intuitive interface for interacting with log data. However, this approach can lead to an unstructured querying experience, as the scope and effectiveness of queries are heavily influenced by individual developers' vocabularies and contextual understanding. This variability can result in inconsistent query formulation across different team members, potentially overlooking critical

information. Moreover, while developers may excel at formulating queries, they often struggle to identify what specific information to query, especially when dealing with unfamiliar systems or services[65]. This challenge is particularly acute in microservice architectures, where developers may lack comprehensive knowledge of all components and their associated logging conventions. A significant limitation of the NL2LogQL models is their tendency to generate queries based on log lines present in the user's input rather than the actual log files. This mismatch can lead to LogQL queries that fail to capture the true semantic intent of the developer's question, resulting in irrelevant or incomplete results. These observations underscore the need for a fundamental redesign of log storage and querying frameworks. The goal is to create a system capable of performing semantic search on log data, bridging the gap between developers' natural language inputs and the diverse, often inconsistent logging practices across different services and teams.

1.1.3 Sauron: Full-fledged semantic search

In the final chapter of this thesis, we introduce Sauron, a semantic search engine designed to address the challenges of querying and analyzing logs in modern microservice architectures. The chapter highlights how traditional log analysis methods face significant limitations, particularly the disconnect between developers who produce logs and engineers who analyze them. In microservice environments, this problem is exacerbated as teams work on isolated components and may not understand the terminology used in logs generated by other services. When services are updated, developers often rewrite log lines using different terminology than the original, creating inconsistencies that make searching across logs difficult. A Microsoft study[65] revealed that 42% of engineers spend over an hour analyzing unfamiliar logs during incidents, highlighting the real-world impact of this problem. Current log search solutions primarily rely on full-text search capabilities that require exact keyword

matching rather than understanding semantic relationships. This approach is inadequate when developers use different phrases to log similar events (e.g., "Authenticated Password" vs. "Password Accepted"). As systems grow in complexity and team sizes increase, maintaining awareness of all logging conventions becomes unsustainable. SAURON addresses these challenges through a sophisticated dual-phase architecture. The indexing phase processes log data to generate vector embeddings that capture semantic meaning, storing them in a specialized vector database. The querying phase employs a modular Retrieval-Augmented Generation (RAG) framework that transforms natural language queries into the same embedding space as indexed documents, performing approximate nearest neighbor searches to find semantically relevant logs. The system includes a query planner that identifies application-specific metadata and rewrites queries to better align with log contexts. This approach significantly reduces the computational burden by narrowing the search space before initiating resource-intensive vector similarity operations. Retrieved logs are passed to an LLM alongside the original query to generate comprehensive, contextually-aware responses. Evaluation results demonstrate that SAURON significantly outperforms traditional methods. A fine-tuned embedding model showed substantial improvements over the base model across three different applications (HDFS, OpenStack, and OpenSSH), with NDCG@10 scores improving by 46.7% to 116.7%. In end-to-end testing, SAURON maintained consistent performance across both base and semantic queries, while comparative systems experienced substantial degradation with semantic variations.

SAURON represents a paradigm shift in log analysis by enabling direct natural language querying capabilities, eliminating technical barriers associated with traditional systems. Its modular architecture provides a flexible framework that can be extended to address specific organizational requirements while democratizing log analysis capabilities across varying expertise levels.

Chapter 2

Bridging the gap in Microservice testbeds

2.1 Introduction

Microservices architectures, first developed to enable organizations to massively scale their services [15], are quickly becoming the *de facto* approach for building distributed applications in industry. Today, major organizations including Microsoft [13], Facebook [123, 125], Google [7], and Etsy [101] are built around microservice architectures.

As microservices grow in importance and reach, the academic study of microservices has similarly flourished. Though the basic principles of the microservice architectural style—that applications should be designed as loosely-coupled, focused services that each provide distinct functionality and interact via language-agnostic protocols [23, 1]—are well-known, there are many open questions around how developers can best design, build, and manage microservice-based applications [57]. For instance, migrating a monolithic application to a microservice architecture is currently a complex, drawn out process [31], as developers must decide on a multitude of factors including (but not limited to) how to determine services' scope and granularity, how to manage message queue depths, and what communication protocols to use. There is no clear guidance, in any domain, to make these choices.

Researchers have conducted a host of user studies with practitioners in the industry to increase the community's understanding of microservice architectures [31, 50, 28]. Independently, the systems community has developed myriad testbeds [2, 55, 138, 156] for evaluating microservices research. Although these testbeds were originally developed to improve or evaluate specific microservice characteristics (*e.g.*μSuite was developed for analyzing system calls made by OLDI Microservices), they are now being used to evaluate a range of research on microservices [73, 55, 70, 99, 51] despite a general understanding that the testbeds' designs are very narrow compared to industry practices.

Over time, the practical deployment of microservices has diverged further from what existing microservice testbeds are able to represent [122]. This mismatch

extends to both testbeds developed by researchers and those developed by industrial practitioners because microservice architectures developed at different companies are proprietary [55, 138]. Research efforts targeted to microservice-based applications risk being useful to only a small set of narrowly-defined (or ill-defined) microservice designs.

The goal of this paper is to provide systematized descriptions of the design axes academic testbeds are built around and how these axes compare to industrial microservice designs. Our systematizations will provide better understanding of the mismatch between testbeds and actual usage of microservices. They will allow for better translation of research results into industry practice, create more awareness of the diversity of microservice implementations, and enable more tailored optimizations. Ultimately, our systematizations will aid the systems community in developing more representative microservice testbeds.

We pair a parameterized analysis of seven popular testbeds, including topological characteristics of the overall microservice architecture, the communication mechanisms used, and whether individual microservices are reused across applications, with semi-structured interviews with microservice developers in industry. Our interviews probe how existing testbeds' design choices are too narrow. They also explore features missing from testbeds that are discussed in the literature to identify their importance for future testbeds. Finally, we contrast the results of our semi-structured interview with the microservice testbeds, culminating in a set of recommendations to guide the designers of the next generation of microservice testbeds.

We find that existing testbeds do not represent the diversity of industrial microservice designs. For example, we find that individual industry microservice architectures use a heterogeneous blend of communication protocols (RPC, HTTP) and styles (synchronous, asynchronous). We also find that industrial microservice architectures vary greatly in the degree to which individual services are reused amongst different

applications or endpoints of the same application. In contrast, testbeds exhibit little to sharing.

We find that participants were unsure of topological characteristics of microservice architectures. Many claimed dependencies among microservices would always form a hierarchy, then admitted this need not be the case. We were surprised to find that a number of participants agreed that service-level cycles could occur, with one service calling another and that service calling the original service. In contrast, the testbeds' dependency diagrams are always hierarchical and do not exhibit cycles.

We present the following contributions:

1. *Systematization of Design Choices:* We systematize the design choices made by seven popular microservice testbeds [55, 156, 138, 122, 2] (Table 2.1). Our systematization provides guidance to researchers about which testbeds are best suited for their work.
2. *Systematization of Industry Microservice Designs*: We expand our design table to include design choices used in industrial microservices (Table 2.3). We use semi-structured interviews with 12 industry participants to collect this data. We collect quotes from our participants to gauge their attitudes about the importance of various microservice design options. We perform our own user study to best encapsulate the most current trends in microservice deployments, and avoid biases from studies that do not distinguish between industrial and experimental microservices [133, 141, 128, 83]. To our knowledge, there is no existing user study that contrasts existing microservice testbeds with industry practices.
3. *Recommendations for Creating New Testbeds*. We present recommendations for improving microservice testbeds by contrasting our systematizations of testbeds design choices with that of industry design choices.
4. *Description of Future Directions*: Through our conversations and analysis of various

academic testbeds, we provide a summary of the current state of microservice design, the discrepancies between testbeds and practice, and recommendations for how to reunite the academic and industry arms of microservice research.

Aspect	DSB-SN	DSB-HR	DSB-MR	TrainTicket	BookInfo	μSuite	TeaStore
Protocol	Thrift	gRPC	Thrift	REST	REST	gRPC	REST
Style	Sync/Async	Sync	Sync	Async/Sync	Sync	Async/Sync	Sync
Languages	C/C++	Go	C/C++	Java, etc.	Multi-lang.	C++	Java
# Services	26	17	30	68	4	3	5
Structure	Hierarchy*	Hierarchy	Hierarchy*	Mixed*	Hierarchy*	Hierarchy*	Hierarchy
Versioning	No	No	No	No	Yes**	No	No
Tracing	Jaeger	Jaeger	Jaeger	Jaeger	Jaeger/Zipkin	None	Jaeger
Testing	U,L	U,L	U,L	U,L	L	L	E2E,L
Security	TLS	TLS	TLS	None	Istio-TLS	None	Istio-TLS

Table 2.1: Summary of microservice testbed design choices. *Indicates partial hierarchy; **BookInfo includes multiple versions for testing. U=Unit Testing, L=Load Testing, E2E=End-to-End Testing.

2.2 Motivation

Microservices is an architectural style wherein a large scale application is built as individual services (called microservices) that work together to achieve a business goal. Figure 2.1 shows two major architectural styles used for building an E-Commerce Application (Business Use Case). The monolithic architecture has multiple functionalities built into a single deployment unit which interfaces with the database deployments to retrieve data to be served. However, in a microservice application, the business use case (E-Commerce), is realized using multiple individual parts - Authentication, Cart, Payment, Product, and User. These individual parts are called "services". They are built to process specific parts of the business domain, and may have their own storage mechanisms wherever necessary instead of depending on a centralised database [136, 135].

The term "microservices" is credited to a 2011 presentation by Netflix [15, 159]. 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 [60]. To circumvent

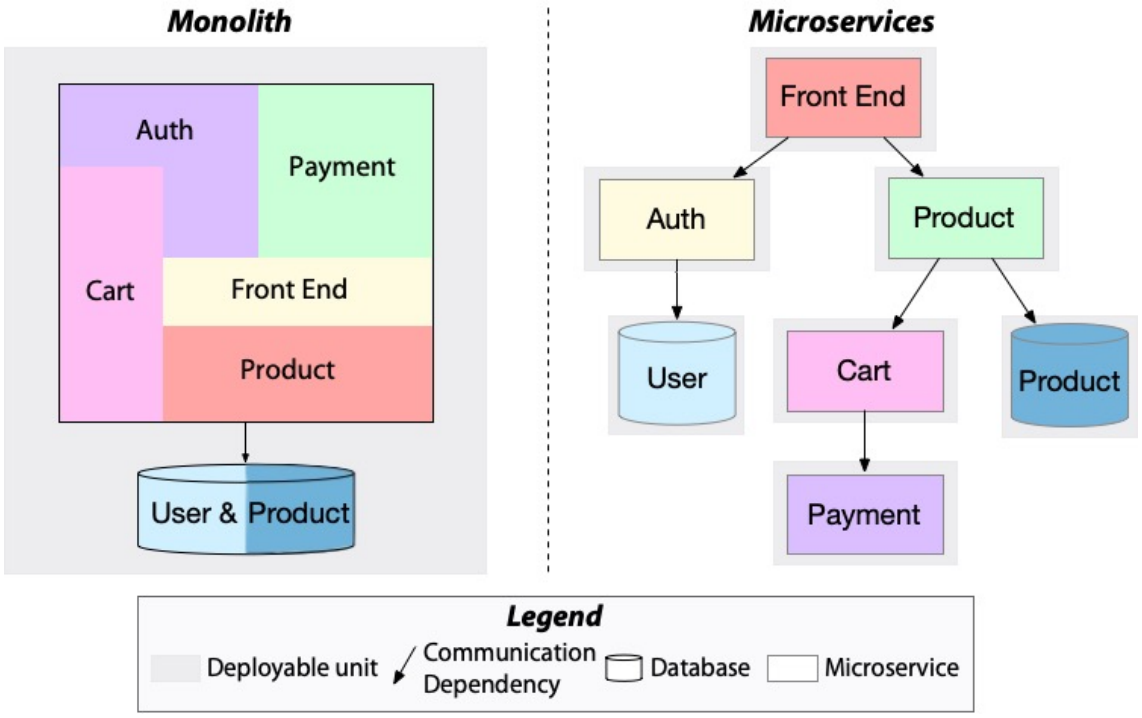


Figure 2.1: **Monolith *vs.* Microservices** A monolith is a single deployable unit, as illustrated on the left. A microservice architecture, shown on the right, is composed of multiple deployable units that communicate with each other.

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[14]. 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 [127, 50, 29, 41, 61, 139, 45, 80]. These extensive studies produced insights on how multiple organizations handled various design choices such as service boundaries, cost of re-architecture and 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 [87, 18, 36, 143] 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 [133, 120, 35, 142, 28, 155, 156, 46, 38, 127, 128, 126, 150]. Of particular note, Wang *et al.* [141] 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.

2.2.1 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.

DeathStarBench Gan *et al.* [55] 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.

TrainTicket Zhou *et al.* [156] 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.

BookInfo he BookInfo benchmarking suite, as a part of the Istio Service Mesh project, serves as a valuable resource for showcasing the prowess of deploying microservice applications using Istio. The benchmarking suite functions as a testbed application that effectively mimics a typical single catalog entry of an online book store. Comprising four distinct microservices, BookInfo is a practical and illustrative example of how Istio can be leveraged in a real-world scenario.

- Product Page: This is the entry point of the BookInfo application. It handles incoming user requests and serves as the user interface, displaying book information. However, it doesn't directly store or retrieve book data. Instead, it relies on the other three microservices for that purpose.

- Details: The Details microservice is responsible for providing in-depth information about a specific book. When a user accesses the product page and requests detailed information about a book, the Product Page microservice delegates the task to the Details microservice.

- Reviews: The Reviews microservice manages user reviews and comments for the books in the catalog. Users can post and read reviews through this service. Like the Details microservice, the Product Page can request reviews for a particular book from this microservice.

- Rating: The Rating microservice is in charge of managing the book ratings, typically in the form of star ratings. When a user views a book in the catalog and wants to see its rating, the Product Page microservice communicates with the Rating microservice to obtain the necessary data.

The synergy between these four microservices is a key element of the BookInfo benchmarking suite's architecture. The Product Page microservice acts as the orchestration layer, coordinating requests from users and gathering information from the Details, Reviews, and Rating microservices. This collected data is then presented to users in the form of an HTML page, offering a seamless and unified user experience.

Overall, the BookInfo benchmarking suite not only serves as a compelling example of microservice deployment but also as a valuable tool for understanding Istio's features and capabilities in managing and securing microservice applications. This practical demonstration helps developers and system administrators grasp the power of Istio in

a real-world context, making it a valuable asset for those exploring microservices and Istio's role in orchestrating and managing them.

μSuite Sriraman *et al.* [122] 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.

TeaStore Kistowski *et al.* [138] 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 [151, 58, 73, 84, 109, 70, 99, 39]. The next most widely used testbed is TrainTicket [109, 156, 155], and the other testbeds are used less commonly in the academic research community.

2.2.2 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 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 [12, 8, 6]. 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.

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.

Style

In microservices architectures, communication patterns can be either synchronous or asynchronous, each serving different purposes and use cases. In synchronous communication, services directly interact with each other through methods like REST or gRPC, waiting for immediate responses before proceeding with their operations. Asynchronous communication, on the other hand, utilizes message queues or event-driven architectures where services can continue their operations without waiting for responses, making it ideal for long-running tasks or scenarios where immediate responses aren't critical. This flexibility in communication patterns allows microservices to be designed for optimal performance and reliability based on specific business requirements.

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.

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.

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[1]. 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,

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

yield important insights, as shown in Alibaba's large-scale traces of microservices [92]. 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.

Number of Services μSuite [11] has 4 distinct applications, each of which have only 3 distinct services [2]. BookInfo has 4 distinct services each deployed as a container within Istio Service Mesh [10]. TeaStore has 5 distinct services with a Registry Service that keeps track of the total number of services in the application [19]. TrainTicket [20] has 68 services including the databases which are deployed as separate containers. DSB-SN[5] has 26 individual containers including the databases and caches, DSB-HR [3] has 17 individual containers including the databases and caches, and DSB-MR [4] has 30 individual containers including the databases and caches.

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.

[2]We derive this number from the installation script provided by the authors in their code [11]

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 [93, 9]. For example, as shown in Figure 2.2, 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, 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.

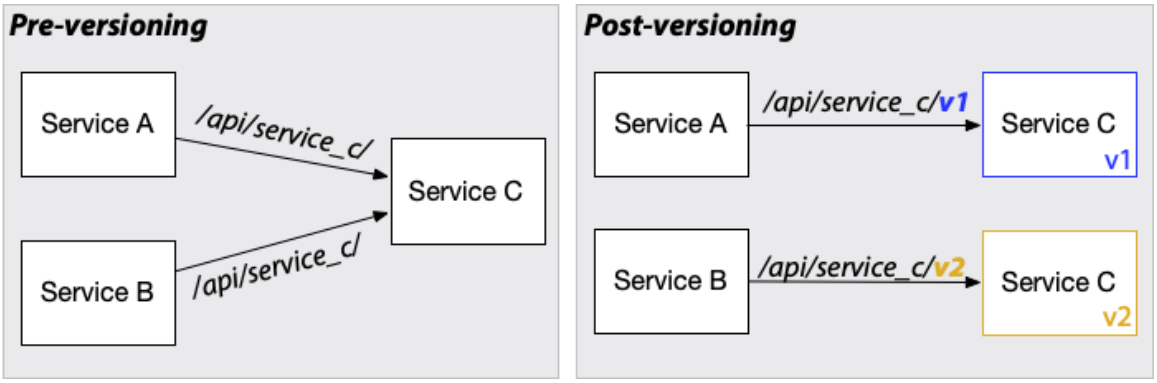


Figure 2.2: **The Versioning Problem:** one approach to maintaining multiple versions of a service is by using versioned APIs.

Versioning Support Only BookInfo provides multiple versions of a service in its testbed. The Reviews Service within the BookInfo application is a notable example, as it comes in three distinct versions. Two of these versions of the Reviews Service access the Ratings Service to fetch and display the book ratings on the webpage. This approach of offering multiple service versions allows users to test and evaluate different

implementations of the Reviews Service within the same environment, providing valuable insights into how changes to a service can affect the overall performance and user experience.

In contrast, other testbeds typically do not explicitly provide pre-configured multiple versions of their services. However, these testbeds often feature extensible APIs that users can leverage to program and deploy multiple versions of a service as needed. This flexibility grants users the freedom to experiment with various service versions, customizations, and configurations, tailoring the testbed to their specific requirements and research objectives. By empowering users to create and manage multiple service versions through extensible APIs, these testbeds facilitate a more dynamic and adaptable testing environment, where the behavior and performance of services can be customized and fine-tuned to suit different scenarios and use cases.

In summary, while the BookInfo benchmarking suite stands out for its provision of pre-configured service versions, other testbeds prioritize flexibility by offering extensible APIs that allow users to program and deploy multiple service versions as desired. Both approaches have their merits, catering to different user preferences and research needs in the realm of microservice testing and evaluation.

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 [111, 26].

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.

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[27] 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.

Security

Microservices architecture presents unique security challenges due to its distributed nature and complex service interactions. Individual services are particularly vulnerable since they often run in containers that may contain unpatched software vulnerabilities, with studies showing over 92% of container images having such vulnerabilities. Inter-service communication security typically employs two primary methods: encryption (SSL/TLS) and inter-service access control mechanisms. However, a significant security concern arises from the traditional design approach where microservices completely trust each other, meaning a compromise of a single service could potentially compromise the entire application. This trust model creates an attack surface where adversaries, after compromising one service, can perceive other services in the network and their exposed APIs, potentially initiating unauthorized requests. To mitigate these risks, modern approaches leverage Software Defined Networking (SDN) capabilities to monitor complex network interactions and enforce security policies. Additionally,

attribute-based access control (ABAC) has emerged as a prominent choice for fine-grained authorization, though the distribution of attributes among ABAC components remains a critical consideration for robust authorization in microservice environments. These security measures must be implemented while maintaining the inherent loose coupling and distributed nature of microservices.

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.

2.3 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 2.3.

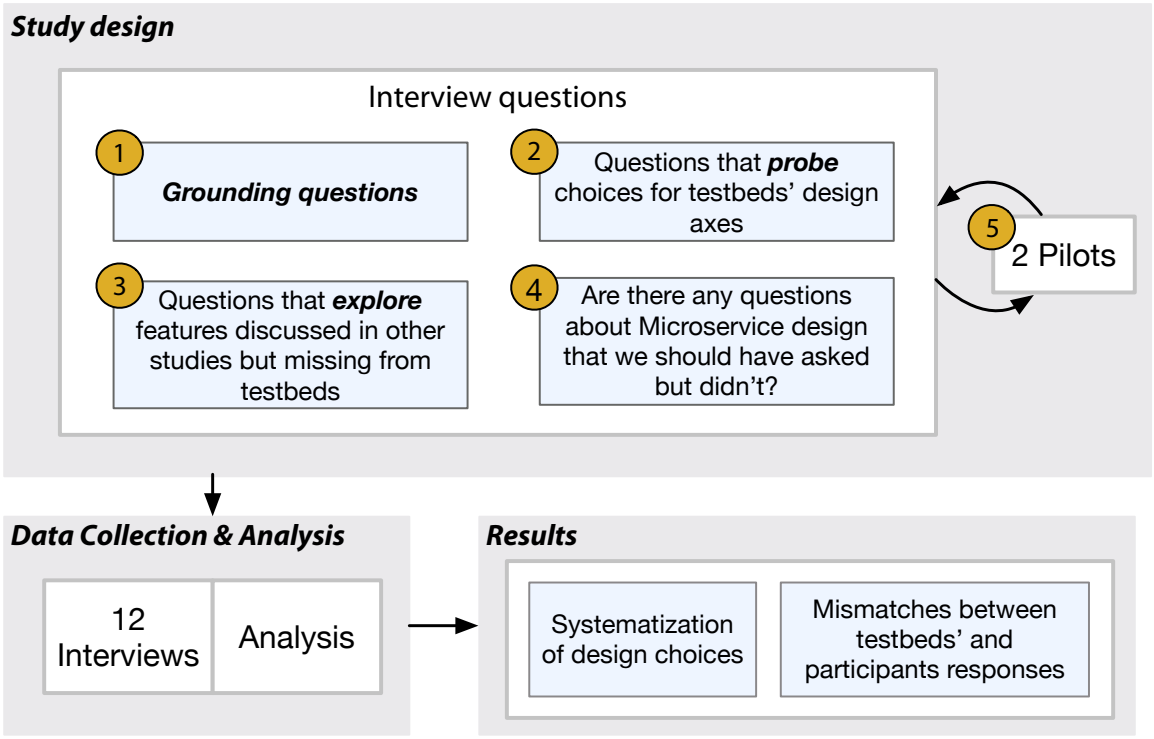


Figure 2.3: **Methodology:** The interview process starts with study design, followed by data collection & analysis, and ends with our results.

2.3.1 Recruiting Participants

We recruited participants from different backgrounds, aiming to collect various perspectives of microservice design choices. 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 [132, 141] to recruit additional participants. We

ID	Skill level	YoE	Sectors worked	Current role
P1	Advanced	10	Government	Full Cycle
P2	Intermediate	3	Finance, Tech, Government	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 2.2: **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.

recruited fourteen participants in total, including the two pilot studies (see below).

Table 2.2 shows demographics of the participants we recruited for our interviews. The table shows that out of the 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.

2.3.2 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. The questions span four categories, described below.

1. **Grounding questions**: These questions ask participants to define microser-

vices 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 [55, 156, 138, 45, 141, 38, 127, 128, 126, 150].

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 2.1 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 [141, 83, 133, 134]), 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 [92], 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 [54, 131, 153]). 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 2.4. (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.

2.3.3 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 2.4 may not always be 12).

2.3.4 Systematization & Mismatches

Systematization: We used the responses to our questions to expand the testbed design axis table presented in Table 2.1 and create Table 2.3. 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.

2.4 Results

Table 2.3 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 2.3. 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 2.2.

2.4.1 Grounding questions

Participants' responses were similar to results in existing user studies [141, 120] and other academic literature [55, 156, 138]. 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 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.

2.4.2 Communication

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."

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.

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 2.3 shows the most commonly used languages among our participants' organizations: Java, Python, C\C++, and Go.

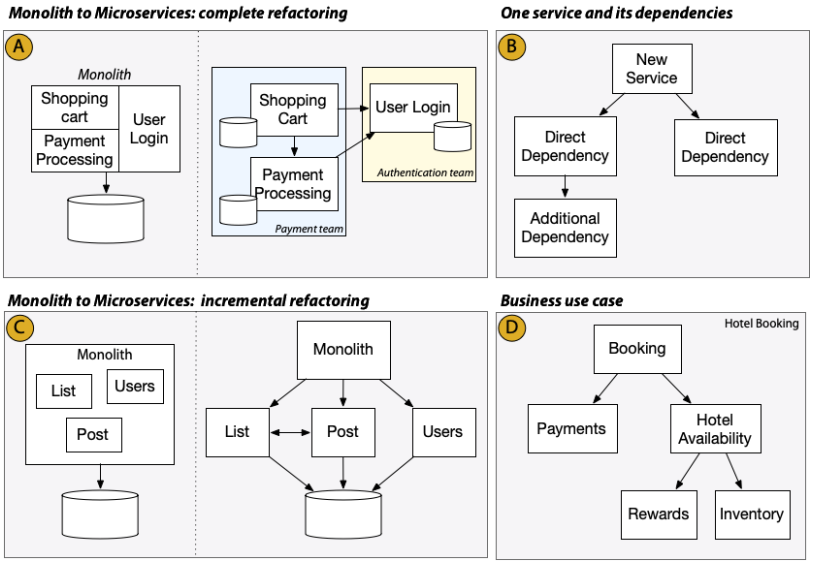


Figure 2.4: **Topology Approaches** For the most part, participants used one of four approaches when asked to draw a microservice dependency diagram that would be used to explain microservices to a novice. Note that (C) represents a hybrid deployment retaining some monolithic characteristics.

2.4.3 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 2.4 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, Ⓐ). 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, Ⓒ). The third approach was to take one service and expand the architecture by considering its dependencies (4/15, Ⓑ). 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, Ⓓ). The single other approach, which is not included in the figure, was centered on container orchestration (P4).

Number of Services We have 12 interpretable responses for the number of services in the applications managed by participants' organizations. As shown in Table 2.3, 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 cant [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 cant tell you who calls what, because it might

depend on the call and it could change day to day."

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]."

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."

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?"

2.4.4 Service Reuse

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."

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.

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."

2.4.5 Evolvability

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 2.3, the methods of versioning support used by the participants include versioned APIs (2/6), explicit support like UDDI (1/6)[17] 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."

2.4.6 Performance & Correctness

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.

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 2.3, 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."

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."

2.4.7 Security

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 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."

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 2.3: **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.

2.5 Analysis

2.5.1 Recommendations and Analysis

The interview results we present in Section 2.4 illustrate that there are a series of gulfs between the assumptions under which testbeds (§ 2.2.1) are designed and the expectations and needs of users and architects in production-level microservice deployments. Following the key design considerations outlined in Table 2.3, 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 2.4.

	DSB - SN	**DSB - HR**	**DSB - MR**	**TT**	**BI**	**MSuite**	**TS**
Comm.							
Style	Both	Sync	Sync	Async	Sync	Both	Sync
Topology							
Cycles	No	No	No	No	No	No	No
Boundaries	BUC, STO	BUC, STO	BUC, STO	DF	DF	3 Tiers	Perf
Service Reuse							
Within App	Yes	Yes	Yes	Yes	Yes	No	No
Across App	No	No	No	No	No	No	No
Storage	In Some	In Some	In Some	In Some	In Some	None	Dedic
Perf.							
SLA for	Supp	Supp	Supp	Supp	Supp	Supp	Supp

Table 2.4: **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. TT = TrainTicket, BI=BookInfo, TS=TeaStore. Supp=Supported

2.5.2 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.

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* [12, 8, 6]. 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 [24, 25]. Thus, an application with a mixture of these protocols would help us measure and mitigate effects of various protocols on resource utilization, latency, etc.

Style

The style of communication impacts the performance of the application, with asynchronous services having higher throughput than synchronous services [124, 140]. 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.

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

2.5.3 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 [81, 102] [3].

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 [156], 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

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

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 [156, 92, 74]. 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.

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.

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 [92]. 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 [92]), 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.

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 [81]. 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 [123, 92, 74, 34, 63], 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.

2.5.4 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.

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.

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.

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 [118, 100]. There is also literature that has explored the field of caches for

microservices, for example placing caches based on workloads experienced by each service [72].

2.5.5 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.

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.

2.5.6 Performance Analysis Support

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 [109, 151, 56, 58] and TrainTicket [109, 155], and these papers tested various methods to scale resources for individual services. While FIRM[109] 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.

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.

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 [109] 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[27] 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 [21][4] 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.

2.5.7 Security

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 [134, 106]. 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

[4]Diffy was archived on July 1 2020.

the service boundaries increase, there is a need for literature on threat assessment for microservice applications.

Chapter 3

Chatting with Logs

3.1 Introduction

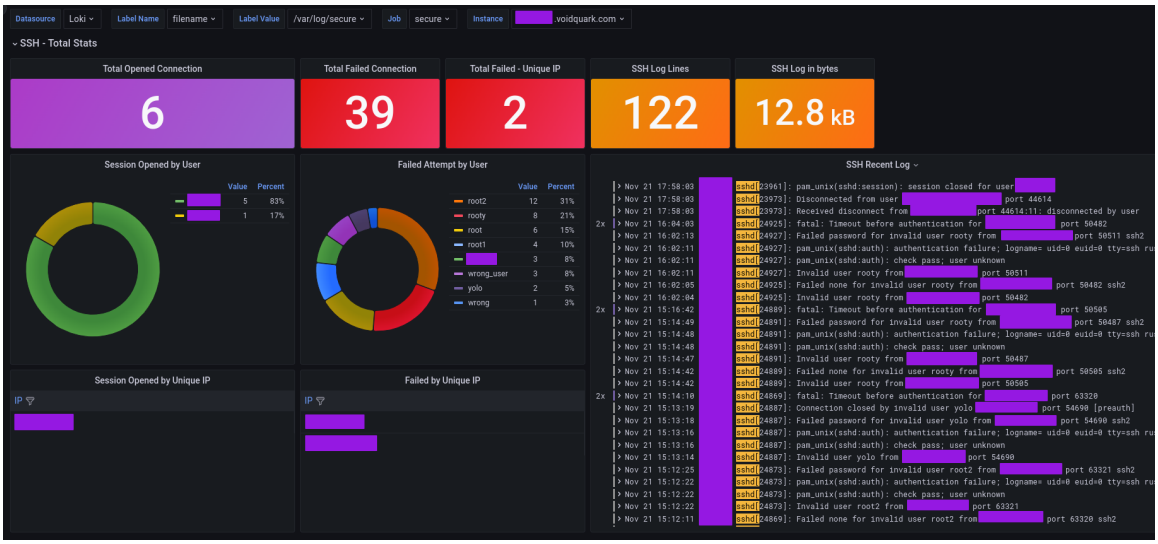
As modern web and mobile applications are increasingly deployed as microservices, observability data (*e.g.* Metrics, Logs, Events, Traces, etc.) are collected from various applications during the application runtime, and the tooling used to collect and store this data increasingly plays a critical role in modern cloud deployments of systems [52]. Observability tools are crucial for understanding how the systems work at scale as they provide insight into key tasks including predicting resource requirements [108, 109, 95], diagnosing faults [58, 79, 88, 157], identifying security breaches [33], and performing regular system checks [116]. Despite the importance of observability tooling, there is a lack of standardization of how a user interacts with the different tools [104].

Log data is typically collected and accessed through various proprietary platforms, each of which have their own query language [104]. Querying this data is invariably *ad hoc* and challenging, involving exact matches to a log format, and keywords to search over a large database [77, 75, 119]. Moreover, there is little to no syntactical overlap between these languages, necessitating significant developer retraining when moving between products.

Logs are often used to power insights dashboards. For example, consider a Grafana dashboard for the OpenSSH application shown in Figure 3.1. This dashboard comprises multiple panels that display information about the application's current state. The first panel in Figure 3.1a shows the total number of open connections, while Figure 3.1b displays the LogQL query required to generate this metric. To construct the query, developers must be familiar with `label_names` and `label_values`, as well as the exact log line syntax, including elements such as "`sshd[`" and "`: session opened for`", with precise attention to whitespace formatting. These components are human-generated without standardized guidelines, which makes it harder for developers to write LogQL queries, as the developers do not have complete knowledge of the various log formats, keywords, labels and other components required to compose these queries.

The lack of standardization extends beyond individual log lines to the various log line formats that developers must handle. Current log parsers, including LLMParser [94] and Drain [64], achieve between 50% to 90% parsing accuracy across applications, presenting an ongoing challenge in the field. The absence of a standardized approach for log line composition results in temporal shifts in formats and syntactical elements required for query construction. These changes in log line structure create difficulties for developers, even those proficient in query languages, as they attempt to formulate and maintain queries over time.

Since various developers face challenges in composing the query, there is a need for a standard query interface that enables developers to write queries more easily. Given the recent advancements in Large Language Models (LLMs)[152], specifically code generation models for languages like SQL [107, 110], we propose that a Natural Language interface will allow developers easy access to observability data by providing a natural language interface over the underlying query language [97]. Generating log queries with an LLM is non-trivial as there are more than 50 different platform



(a) Grafana Dashboard for OpenSSH logs retrieved from [137]

```
sum by(instance) (count_over_time({
$label_name=~"$label_value", job=~"$job",
instance=~"$instance"} |="sshd[" |=":␣session␣opened␣for"
| __error__="" [$__interval]))
```

(b) LogQL Query for calculating the total number of open connections (First panel)

Figure 3.1: Example Grafana Dashboard

specific log querying languages, and each model offers varying degree of support for them. LLMs require a fundamental understanding of the target query language to generate effective queries. Without this knowledge, LLMs may either refuse to generate queries or, in more problematic cases, produce semantically or syntactically incorrect queries [68]. The LLMs often end up generating queries that are non-executable and in situations where the LLMs know the query language, efficiently prompting them is often not enough to generate realistic queries (as we demonstrate in §3.2.3) due to insufficient knowledge of log lines which are too big to be fit into context window of the application. On the other hand, finetuning the LLMs for various languages requires rich set of natural language questions, corresponding queries and the output from the queries to serve as ground truth to test the efficiency of the model.

The challenge of querying logs with natural language exhibits significant similarities to problems encountered in data analysis, such as text-to-SQL conversion. Composing queries for log searches frequently involves the utilization of both structured elements, such as labels [121] and tags [47], as well as unstructured components from log entries that need to be included or excluded. In both scenarios, there is often a disconnect between the individuals generating the data (*e.g.* tables, rows, logs) and those querying it for information extraction [65]. However, these problems diverge in terms of data characteristics and scale. While SQL tables may have varying schemas, the diversity in log line formats is considerably more extensive. For example, common text-to-SQL benchmarks like SPIDER [149] contain 5 tables per database, whereas each application in LogHub [158] contains more than 25 log formats per application.

In this work, we take the first steps towards building realistic NL interfaces for generating log queries. To achieve this, we first create a dataset of natural language to LogQL queries and use it to fine-tune a suite of popular LLMs creating LogQL-LM, a system designed to convert natural language questions into LogQL queries. LogQL is a specialized query language for searching and analyzing log data within Grafana's open-source log aggregation system, Loki [82]. The selection of LogQL was motivated by its open-source nature and the extensive availability of Grafana dashboards (Fig 3.1a), which enable the formulation of realistic natural language queries, and its support in various open source LLMs.

Through our exhaustive evaluation, we demonstrate that fine-tuning on our dataset significantly enhances the performance of popular LLMs, including GPT-4o, Llama 3.1, and Gemini, in generating accurate LogQL queries. Our experiments reveal that GPT-4o achieves up to 75% and 80% improvements in accuracy and F1 respectively, with fine-tuning enhancing query outputs, reducing syntax errors, and improving label matching and temporal aggregation. We perform further ablation studies to evaluate the effects of the number of training examples and the potential transferability of

these models across applications.

Specifically, this paper makes the following contributions:

- First, we present and release a dataset**NL2LogQL**designed to facilitate the development and benchmarking of natural language to LogQL systems, with a particular focus on fine-tuning Large Language Models (LLMs) to generate syntactically and semantically correct LogQL queries. NL2LogQL consists of 424 manually curated natural language to LogQL pairs. Each pair is derived from a panel in a Grafana Community Dashboard, covering three distinct applications. The dataset was constructed by manually describing the purpose of each panel in natural language and crafting the corresponding LogQL query. This resource represents the first dataset specifically designed to enable an NL-to-LogQL interface.

- Second, we present a web-based interface that enables developers to generate LogQL queries for the aforementioned applications. This interface serves a dual purpose: it provides a practical tool for query generation and acts as a platform for collecting additional natural language questions, thereby facilitating the continuous expansion of the dataset.

- Third, utilizing this novel dataset, we fine-tune three off-the-shelf LLMs for the task of natural language to LogQL query generation. We release these fine-tuned NL-to-LogQL models, along with the prompts used for their evaluation and a fine-tuned CodeBERTScore model for assessing the results.

- We establish a set of metrics to quantitatively assess both the syntactic and semantic correctness of LogQL queries generated by LLMs, and further conduct a comprehensive study on the efficacy of fine-tuning models, analyzing the optimal number of samples, the impact of post-fine-tuning prompting, and the transferability of models across various applications.

3.2 NL Interface for Log Search

In this section, we present the challenges associated with querying logs, highlight specifics of an open-source log query language called LogQL, and present an initial sketch of how we can use LLMs to translate natural language queries to LogQL.

3.2.1 Challenge: Querying Logs is Difficult

Existing tools for storing and querying log data to obtain insights present a significant usability challenges for many reasons. We highlight these challenges below.

Steep Learning Curve. These tools require the developers to learn and use esoteric tool-specific query languages, that have a steep learning curve [104]. Due to the unintuitive nature of these languages, users often struggle with constructing effective queries to find specific log entries [43]. Consequently, only a small percentage of power users within an organization can leverage the full capabilities of log analysis tools [42], hindering the democratization of log analysis. Recent studies [48] have also highlighted the challenges faced by new team members in using existing tools, requiring significant time to gain proficiency.

Insufficient Context. In the context of DevOps [144, 116], where the developer and the operator are usually different individuals thus the person writing queries often lacks sufficient context. To construct effective queries, the operators often require detailed knowledge of log lines. Operators often lack appropriate context as they're dealing with unfamiliar logs, or context switching between multiple tools and dashboards [65, 49].

Large and unstructured logs. The complexity of writing these queries increases due to the high volume of application logs and their varying formats. For example, to write the LogQL query in Fig. 3.1b, the developer must know syntax of the log querying language (LogQL) and also the semantics the log file – such as "*sshd[*",

"*session opened for*" – that are required to construct this query.

Due to these inherent complexities, log data analysis remains predominantly within the domain of software developers who possess intricate knowledge of the logging systems. The utilization of log data presents significant untapped potential for informing strategic business decisions [90]. For example, insights from HTTP header parsing can optimize marketing campaigns by understanding regional traffic patterns, and analysis of distributed traces can reveal the most commonly used platform features. Questions like "*what is the average response time for my website?*" or "*what are the most common errors being reported?*" can provide a more detailed picture of system performance and identify areas for improvement. However, similar to data analysis domain [103], non-engineers who want to extract such information must either rely on engineers to write queries or learn the querying language themselves, creating a barrier to data-driven decision-making across the organization.

To write effective log search queries, developers often need to have complete syntactical and semantic knowledge of the query language and log lines. These challenges collectively underscore the need for more intuitive and user-friendly interfaces to log analysis that can improve productivity, accessibility, and cross-functional utility of log data.

3.2.2 Background: LogQL

LogQL is a query language designed for searching and analyzing log data in Grafana's log aggregation system, Loki [82]. Loki focuses on indexing metadata rather than full log text. LogQL provides tools for filtering, aggregating, and extracting insights from log streams, supporting both log and metric queries, with a structure that includes label selectors, line filters, and time range specifications.

LogQL provides users with tools to filter, aggregate, and extract insights from log

streams, making it valuable for monitoring, troubleshooting, and maintaining complex distributed systems. Operators typically write a LogQL query per panel that are then arranged together to form a dashboard (for example, the dashboard in Figure 3.1a). The language supports two primary query types: log queries for retrieving and filtering log content, and metric queries for applying aggregation functions to transform log data into numerical time series.

As Loki's indexing strategy focuses on the metadata associated with log lines, rather than the full log text, this indexing approach has implications for LogQL queries, as they must include the relevant tags to search the indexed metadata effectively. For example, given the log line [2019-12-11T10:01:02.123456789Z {APP="NGINX",CLUSTER="US-WEST1"} GET /ABOUT], Loki will index the timestamp and the labels attached to the log line, such as "app" and "cluster", but not the actual log text starting from "GET". Since the indexing of the logs is based on timestamp, the queries are relative to the current system time. Thus, LogQL provides labels to allow filtering log lines using metadata. Labels are key-value pairs associated with log streams, providing metadata about the logs' origin and characteristics.

Consider the human written query in Figure 3.2b (green colour) to quantify the occurrence of authentication-related service unavailability errors in an OpenStack deployment within the Asia-Pacific region over the past 30 days. It begins with label selectors enclosed in curly braces: {*job="openstack", region="asia-pacific"*}. The *job="openstack"* label identifies logs from OpenStack services, while *region="asia-pacific"* narrows the focus to the Asia-Pacific region. Following the label selectors are two line filters: —= *"503"* and —= *"token validation"*. These filters use the —= operator to perform case-sensitive matches, selecting log lines containing the HTTP status code 503 (indicating a service unavailable error) and mentioning "token validation". The query concludes with a time range specification "*[30d]*", which defines a 30-day analysis window. The "*count_over_time*()" function wraps the entire

log selection criteria, counting the number of matching log lines within the specified time range. The complete list of filters and aggregation commands can be found in LogQL's documentation [62].

3.2.3 Our Vision: LLM assisted query generation

The heterogeneity of log query languages necessitates enhanced query composition interfaces. While existing "query builder" interfaces ensure syntactic correctness, they depend on developers' expertise in log line selection. LLMs have shown efficacy in log-related tasks, leveraging their ability to process unstructured text. However, direct LLM application to log search presents challenges in handling large-scale data, efficient indexing, and real-time capabilities [43].

We propose utilizing LLMs for log query generation, balancing accessibility and efficiency. This approach, applied successfully in SQL generation [97] and data analysis [146], leverages LLMs' natural language understanding to translate search intents into optimized queries. This method bypasses complex indexing requirements while maintaining LLM capabilities, enabling execution through existing log search systems. It reduces the query language learning curve, facilitates faster iteration for experienced developers, and aligns with engineers' mental models of their systems[78].

To adapt LLMs for specific tasks, two lines of approaches have been employed in the past: (i) In-Context Learning (ICL) [40]; and (ii) fine-tuning pre-trained models with task-specific examples[113]. We discuss the potential and limitations of both in generating LogQL queries below.

ICL incorporates task-specific demonstrations into the input during inference, guiding the model without parameter retraining. However, incorporating large log files is impractical for smaller models due to limited context windows [44], and models with larger context windows often exhibit instability and reduced robustness [89]. Similar to SQL generation issues [67], LLMs often generate non-existent log lines. To

demonstrate this, we devised a prompt with documentation, examples, and instructions for generating queries in Datadog Query Language (DQL), LogQL, and `grep`.

Figure 3.2 presents an example query for searching service not found errors during token validation in the past 30 days for a specific OpenStack node. Comparison of LLM-generated queries (red color) with human-crafted (green color) ones reveals semantic inconsistencies across DQL, LogQL, and `grep` methods. In DQL, the LLM-generated query uses non-standard attribute names (e.g., "status.code:503" instead of "@http.status_code:503"), indicating gaps in platform-specific convention comprehension. The LogQL LLM query erroneously employs a non-existent function (e.g., "calculate_over_time" instead of "count_over_time") and misapplies operators (using "!=" instead of "—=" for inclusion). The `grep` LLM query presents a generic structure, lacking context-specific knowledge of log formats (using generic error statements like "error" or "failed" instead of specific log patterns) and misapplying common log analysis patterns (e.g., searching for "token validation" instead of specific token-related log entries). These errors, combined with log lines not fitting into the context window, demonstrate that using ICL based approaches fails to generate realistic log query language queries.

Fine-tuning, particularly few-shot tuning, offers significant advantages for adapting pre-trained LLMs to specific tasks such as LogQL generation. This approach involves re-training the LLM on a tailored dataset, allowing the model to adjust its internal parameters and better align its outputs with desired outcomes. Few-shot tuning enables LLMs to generalize from limited examples, facilitating the extraction of relevant information across diverse log formats and applications. This is particularly crucial given the often *ad hoc* nature of log files, which lack standardized logging procedures. By providing more diverse log examples during the training phase, fine-tuning enhances the model's ability to handle varied log structures.

Prior studies [113, 148] have demonstrated that few-shot tuning offers superior

accuracy at lower computational costs for related tasks like text-to-SQL. Moreover, the efficiency of few-shot tuning, requiring only a small number of data samples, results in a rapid fine-tuning process without significant time overhead.

Importantly, few-shot tuning eliminates the need for continuous in-context demonstrations during inference, potentially reducing overall query latencya critical bottleneck for log queries [77]. This reduction in query latency is especially vital in log search systems, where rapid data retrieval and analysis are essential for real-time monitoring and troubleshooting of complex distributed systems. Faster query times enable IT teams to detect and respond to issues more quickly, minimizing downtime and improving overall system performance.

Few-shot tuning offers a promising approach for adapting LLMs to log query generation tasks. This method enables efficient generalization from limited examples, reduces inference time, and allows for diverse log example training. Given these advantages, we employ few-shot tuning to fine-tune LLMs for LogQL query generation in this study.

3.3 LogQL-LM

We first define the problem of translating a natural language log-query to LogQL as follows.

To train such a mapping $NL2LogQL$, and evaluate its logging efficacy, we first require a dataset that provides us with example tuples $(DB, q^{\text{NL}}, q^{LOG}, a)$ each having a logfile DB, a natural language query q^{NL}, a ground truth correct LogQL query q^{LOG}, and the output of executing the query, a. We manually create such a dataset having 424 example tuples as described in Section 3.3.1. Using this dataset, we describe our approach to fine-tuning existing large language models to automatically map any new natural language query for any new logfile to the intended LogQL query.

NL Query	LogQL	Query Output
How many times did the NameSystem allocate new blocks in the past minute for hdfs-south-america?	sum(count_over_time({application=“hdfs-south-america”}— ∼“BLOCK* NameSystem\\ .allocateBlock:” [1m]))	1880
How many times did PAM ignore max retries in the last 24 hours for openssh-us-east?	sum(count_over_time({application=”openssh”, hostname=”us-east”} —= ”PAM service(sshd) ignoring max retries” [24h]))	39700
Show me the most recent successful login for user ’fztu’ in openssh-asia-pacific, including timestamp and source IP?	{application=”openssh-asia -pacific”} —= ”Accepted password for fztu” — regexp ”(?P<source_ip>\\d+\\.\\d+ \\.\\d+\\.\\d+)”	120 Log lines with Accepted password for fztu
What are the top 3 most frequent exceptions encountered during writeBlock operations in the past 24 hours for hdfs-asia-pacific?	topk(3, sum by (exception_type) (count_over_time({component=∼”dfs.DataNode.*”, application=”hdfs-asia-pacific”} —∼ ”writeBlock .* received exception” — regexp ”writeBlock .* received exception (?P<exception _type>[ˆ:]+)” [24h])))	{exception_type=”java.io. EOFException”} {exception_type=”java.io. IOException”} {exception_type=”java.io. InterruptedIOException”}

Table 3.1: Example tuple from our dataset showing the NL query LogQL query and the corresponding output. The first 2 rows represent metric queries and the next 2 represent log queries

3.3.1 Dataset

As defined earlier, to effectively fine-tune a model to tranform natural language queries to their LogQL counterparts, we require a comprehensive dataset consisting of $(DB, q^{\text{NL}}, q^{\text{LOG}}, a)$ tuples. Table 3.1 shows examples of such tuples for two metric queries and two log queries.

Data Sources. Constructing a dataset of such records necessitates both realistic logs and natural language questions to ensure the model's applicability for operators writing queries on their applications. We source logs from the LogHub 2.0 dataset [158],

which encompasses logs from diverse applications and includes various event extraction templates for log parsing tasks. To obtain realistic NL queries, we analyze the Grafana Community Dashboards [112], as these dashboards are open source and publicly available. We extract NL questions based on panel titles and displayed information. Our analysis of these dashboards informs the dataset construction, involving the creation of LogQL queries corresponding to the presented panels. For panels requiring multi-step queries, such as pie charts displaying failed login attempts by users, we decompose them into separate NL to LogQL entries in our dataset. For example, Figure 3.1a illustrates an example dashboard for OpenSSH, publicly available and comprising 7 metrics panels and 2 log panels, which provides valuable information for developers to establish alerting systems or query data. (e.g., "show the total users with failed attempts" and "how many failed login attempts for $\$username$"). While we utilize Grafana and LogQL for our dataset construction, our approach is extensible to other dashboards and log query languages.

Drawing insights from the construction of analogous datasets for text-to-SQL or data analysis tasks [148, 98, 37], we recognize that dataset diversity is crucial for producing fine-tuned models capable of addressing a wide range of queries. To develop models that are valuable for operators in querying their logs, we have identified three key domains across which we ensure dataset diversity: application type, use case, and LogQL operation complexity. We create a total of 424 individual entries in our dataset, across 3 applications encompassing a wide range of use cases and LogQL operations. We spent 500 hours of human labour between the authors to create and validate the dataset.

Application Diversity Our dataset mirrors the database diversity observed in text-to-SQL tasks, where the models are finetuned to generate queries across different databases. This diversity is essential for log analysis, as each application generates unique log formats, lines, and structures, which are crucial for constructing accurate

LogQL queries. To ensure a representative range, we have built our dataset using logs and dashboards from three distinct applications: OpenSSH, OpenStack, and HDFS. These applications span diverse domains of system operations: OpenSSH facilitates secure network communications, OpenStack manages cloud computing resources, and HDFS enables distributed storage of large data volumes across commodity hardware clusters. For the dataset, we wanted to have at least 50 samples per application, to be consistent with common text-to-SQL benchmarks such as Spider[149] and BIRD[86]. Our dataset comprises 155, 154, and 115 samples for OpenSSH, OpenStack, and HDFS, respectively.

Use case Diversity For OpenSSH, we identified 7 distinct use cases: Suspicious Activities, Brute Force Attempts, Connection Analysis, Invalid User Attempts, System Health and Performance, User Session Analysis, and Authentication Failures. OpenStack presented 11 use cases, including Instance Lifecycle, Audit and Synchronization, Resource Usage, System Health and Maintenance, API Performance, Instance Lifecycle Management, Image and File Management, Network Operations, Security and Authentication, Error Analysis, and API Performance and Requests. HDFS contributed 7 use cases: Replication and Data Transfer, Error Analysis, Performance Issues, Data Transfer and Replication, Performance Monitoring, Block Management, and NameNode Operations. This variety of use cases across applications ensures that our fine-tuned models can address a wide spectrum of log analysis scenarios, enhancing their practical utility for operators.

LogQL Operation Diversity The composition of our dataset reflects this diversity in LogQL operations, as illustrated in Table 3.2. For log queries, 65.8% use single line filters, while 34.2% employ multiple line filters. In terms of label filters, 36.5% of queries use single label filters, and 63.5% use multiple label filters. It is important to note that these percentages are independent; a query with a single label filter can still have multiple line filters, and vice versa. This distribution ensures a

Type of query	Filter(s)	Percentage
Log	Single Line	65.8
	Multiple Line	34.2
	Single Label	36.5
	Multiple Label	63.5
Metric	Log RA	40.1
	Unwrapped RA	7.8
	Built-in RA	40.1

Table 3.2: LogQL Query Types and Filters with corresponding values in our dataset

balanced representation of both simple and complex log query structures. For metric queries, we observe an equal distribution between log range aggregation and built-in range aggregation, each accounting for 40.1% of the metric queries. Unwrapped range aggregation is less common but still represented, comprising 7.8% of the metric queries. This distribution of query types and filters in our dataset provides a robust foundation for finetuning models capable of handling a wide array of LogQL query scenarios.

3.3.2 Finetuning LLMs

In this section, we present the models used for fine-tuning and prompting for the task of *NL2LogQL*.

For finetuning, we require models that can excel at various coding and reasoning tasks and can learn to specifically generate syntactically and semantically correct LogQL queries. To ensure a systematic empirical evaluation across a diverse set of models, we selected three widely recognized models that have achieved state-of-the-art performance in various coding and natural language tasks. Specifically, we employed LLama-3.1 [130], Gemma-2 [129], and GPT4o [105], for subsequent fine-tuning of *NL2QL*. Each model has unique strengths, especially in reasoning-heavy tasks like Measuring Massive Multitask Language Understanding (MMLU) and coding benchmarks [66].

- `GPT4o` is robust in both reasoning and coding, especially for complex queries,

but it's closed-source and proprietary.

- `Gemma-2-9B` is the smallest LLM of the Gemma-2 series, released by Google, which, despite its size, offers competitive performance due to improved parallelized training. This model balances efficiency and performance, making it ideal for scalable applications like *NL2LogQL*.

- `LLama-3.1-8B` is another notable LLM, leveraging Grouped-Query Attention (GQA) for improved inference scalability. GQA enables more efficient handling of large input sequences, which is critical for inference.

We fine-tune these models using LoRA [69], which introduces a small set of additional trainable parameters while freezing the original model. By using low-rank parameterization, LoRA reduces computational and memory costs, enabling faster convergence with minimal performance impact. **Training Hyperparameters**: We used the AdamW optimizer [91] with 8-bit quantization, implemented via the bitsandbytes library. The learning rate was set to 1e-4, following a cosine schedule with a warm-up of 10 steps. The training was run for 4 epochs with a micro-batch size of 4 and gradient accumulation steps of 1. To optimize memory usage, we applied gradient checkpointing, 8-bit quantization for the base model, and flash attention. LoRA parameters include a rank of 16, a scaling factor of 32, and a dropout rate of 0.05. All our models were trained on 1 Nvidia A100 GPU available via Modal Labs [85]

3.3.3 Metrics

Similar to prior work in text-to-SQL [114], we evaluate the performance of the our finetuned models using Exact Match and Execution Accuracy. To assess Execution Accuracy, we compared the results returned by the LogQL queries generated by our models to the results from manually written reference LogQL queries. Since there are

two main types of LogQL queries, Metric and Log, we used different metrics tailored to each.

- For **Metric queries**, which return a numerical value, we compared the model's output and expected output. We find the output accurate if it is exactly the same as the expected output, and any deviation is considered a wrong output, for floating point outputs we compare up to two decimal places rounded up.

- For **Log queries**, which return a list of relevant log lines, we computed the precision and recall of the model's output log lines compared to the reference log lines. We calculate the F1 score, which is the harmonic mean of precision and recall, as a summary metric. These metrics evaluate how well the model's queries are filtering the logs to surface the most pertinent information. Since LogQL results are always sorted by timestamp, we did not need to compare the relative ranking of the model and reference outputs - the ordering is guaranteed to be consistent as long as the same logs are returned.

Along with these metrics, we also make use of Perplexity Score [147], to assess the language model's ability to predict the next token in a sequence, providing a measure of how well the generated code aligns with the model's learned probability distribution. A lower perplexity score indicates that the model is more confident and accurate in its code predictions.

While comparing the output of the model is meaningful for checking the end result, it doesn't account for the syntax lapses causing the outputs to be dramatically different. Relying solely on output-based evaluation methods can be misleading, as they may fail to capture the underlying issues in the generated LogQL queries. To address these limitations, we assess the exact match between the generated query to ground truth query.

While Perplexity scores offers an intrinsic metric for the confidence of a model

in generating accurate LogQL queries, it doesn't provide an extrinsic metric. To mitigate this challenge, we make use of CodeBERTScore(CBS) [154]. CBS is a pre-trained language model specifically trained for evaluating code outputs in various programming languages. By finetuning CBS on a subset of LogQL queries from our dataset, we evaluate the generated LogQL queries based on their semantic similarity to reference queries and their adherence to the syntactic rules of the LogQL language. The CBS evaluates the similarity between generated and reference LogQL queries using cosine similarity, producing values between 0 and 1. A score of 1 indicates perfect semantic and functional equivalence, while 0 represents complete dissimilarity. Scores above 0.7 strongly correlate with high query quality and correctness as judged by human evaluators, whereas scores below 0.4 typically indicate significant functional deficiencies. The metric's relative nature means it is most meaningful when comparing queries within the same evaluation context.

By combining Perplexity Score and CodeBERTScore, we can obtain a more comprehensive assessment of the generated LogQL queries. These metrics provide insights into the model's ability to generate syntactically correct and semantically meaningful code, which is essential for understanding the model's performance and limitations.

3.3.4 Demonstration

To facilitate comprehensive model comparison and enhance user interaction, we developed a web-based interface that enables simultaneous evaluation of multiple LLM responses. As illustrated in Figure 3.3, the interface presents a streamlined design with a prominent query input field at the top, where users can formulate natural language questions about log analysis. Upon submission, the system concurrently processes the query through three distinct fine-tuned models: GPT-4o, Llama-3.1, and Gemma2. The responses are displayed in parallel panels, each showcasing the

generated LogQL query along with its response time. For example, when querying about instance build times in OpenStack deployments, each model generates specialized LogQL syntax incorporating regex patterns and temporal aggregations. The interface displays response times (2.9s, 49.56s, and 67.8s respectively), enabling quantitative performance comparison. This parallel visualization approach not only facilitates direct comparison of query formulation strategies but also provides valuable feedback mechanisms for continuous model improvement. The comparative layout effectively highlights the nuanced differences in how each model interprets and translates natural language queries into LogQL syntax, contributing to our understanding of model behavior and performance characteristics. The demo is hosted on https://llm-response-simulator-alt-glitch.replit.app

3.4 Evaluation

To validate our natural language interface for LogQL query generation, we established a comprehensive evaluation framework focusing on the models' capability to generate executable queries that yield accurate results. The evaluation framework covered several key areas: first, a comparison between fine-tuned and baseline models; second, an analysis of how the size of the fine-tuning dataset affects model performance; third, an exploration of how well the model transfers across different application domains; and finally, a qualitative review of the generated LogQL queries, using CodeBERTScore as an objective measure. This comprehensive approach allowed for a detailed assessment of LogQL-LM's robustness and its ability to generalize to different use cases.

Figure 3.4 contains the complete pipeline of our evaluation. To ensure compatibility with Loki's indexing system (§3.2.2), we preprocessed the logs by converting them to Loki format using the log parsing templates provided in the LogHub dataset. Key-value pairs for the various labels were derived from the keys in the log templates,

with corresponding values parsed from each log line using existing parsers. Since Loki performs relative-time querying and indexes data based on timestamp, we converted timestamps in logs to to reflect more recent dates, while maintaining relative ordering of the log lines. Since most log queries are for searching through the logs in the last 7 days [77], this pre-processing step proved crucial for facilitating efficient querying and analysis of the log data.

Natural language questions from our test set are processed through different infrastructures: vLLM for LLama and Gemma models, and OpenAI API for GPT-4o. The generated LogQL queries are then executed on a locally deployed Loki instance, ensuring consistent execution conditions and eliminating network-related variabilities. The responses undergo comparative analysis, measuring various performance metrics as detailed in Section 3.3.3.

3.4.1 Performance of finetuned models

Model	App	MQ (B)	MQ (A)	LQ (B)	LQ (A)	Pplx (↓)
Llama	OSSH	0.03	0.50	0.05	0.42	15.2
	OSTK	0.05	0.45	0.06	0.42	19.8
	HDFS	0.02	0.48	0.1	0.59	22.5
Gemma	OSSH	0.06	0.25	0.11	0.31	38.0
	OSTK	0.02	0.27	0.12	0.47	36.7
	HDFS	0.07	0.35	0.14	0.39	27.7
GPT4-o	OSSH	0.21	0.74	0.16	0.62	9.8
	OSTK	0.28	0.82	0.18	0.68	10.2
	HDFS	0.23	0.79	0.19	0.74	10.7

Table 3.3: Results for models (B)efore and (A)fter finetuning. MQ = Metric Queries measured by Accuracy; LQ = Log Queries measured by F-1 Score; Pplx = Perplexity

In this experiment, we wanted to test the effeciency of the finetuned models in generating LogQL queries compared to the base model. We used 50% of the samples from each application for finetuning the models using the method described in Section 3.3.2. Our experimental evaluation of the NL2LogQL translation models

reveals significant improvements in accuracy and F-1 score through finetuning, as illustrated in Table 3.3. Most of the queries generated were executable, except for 10% of the queries that had wrong syntax such as no log lines after filters or had ill-formed regular expressiones. Among the finetuned models, GPT-4o exhibited the strongest performance, achieving remarkable post-finetuning accuracy scores ranging from 0.74 to 0.82 and F1 scores between 0.62 and 0.74, up from pre-finetuning metrics of 0.21-0.28 for accuracy and 0.16-0.19 for F1 scores. Llama-3.1 showed significant enhancement, with accuracy improving from below 0.05 to approximately 0.50 across applications, alongside F1 scores rising from around 0.05 to the 0.42-0.59 range. While Gemma-2 demonstrated more modest gains, it still showed meaningful improvements, with accuracy increasing from below 0.07 to 0.25-0.35 and F1 scores improving from 0.11-0.14 to 0.31-0.47. The perplexity scores further support these findings, with GPT-4o achieving the lowest perplexity (9.8-10.7), followed by Llama-3.1 (15.2-22.5), and Gemma-2 (27.7-38.0), indicating superior model coherence and predictive capability. Most of the correct responses from the models before finetuning came from providing a lot of context to the model for generating the LogQL query.

Figure 3.5 contains samples of LogQL queries generated by the LLM before and after finetuning the model using our dataset. Before finetuning, the LLM generated queries exhibited several common errors: incorrect label usage (e.g., “app” instead of “application”), syntax errors in timestamp placements, wrong filter specifications, invalid grouping syntax, and improper matching operators. For instance, in Figure 3.5a, the pre-finetuning query for GPT-4 showed incorrect label usage and timestamp placement. Similarly, Figure 3.5b demonstrates Llama’s incorrect use of negative matching operators, while Figure 3.5c shows Gemma’s invalid grouping syntax in `count_over_time` operations.

The fine-tuned models exhibited substantial enhancement in query generation capabilities, producing syntactically valid and executable LogQL queries. For example,

in Figure 3.5a, the corrected query properly implements regexp capture groups `((?P<source_ip>[\\d\\.]+))` for IP extraction and uses correct sum aggregation with proper label matchers. In Figure 3.5b, the finetuned model correctly uses positive matching operators (—=) and proper application labeling (application=“openssh”). Figure 3.5c shows the correct implementation of `count_over_time` with proper time window specification [1h] and appropriate temporal aggregation. These improvements resulted in queries that could accurately capture and format the desired log data while maintaining proper syntax and execution capability.

The experimental results demonstrate that finetuning significantly enhanced the performance of all three LLM models in generating LogQL queries, with GPT-4o showing the most impressive improvements in Accuracy and F1 scores by up to 75% and 80% respectively. Post-finetuning, the models produced 20% more executable queries with fewer syntax errors, improved label matching, and better temporal aggregation, highlighting the effectiveness of the finetuning process in enhancing the models’ ability to generate accurate and functional LogQL queries.

3.4.2 Effect of number of finetuning samples

In the previous experiment, we looked into the effect of finetuning for enhancing the ability of the models to generate LogQL queries compared to the base model. Previous works from other log related tasks [94], and text2sql [97] have shown that the performance of models change based on the number of samples used for finetuning. Constructing the dataset for finetuning these models is an arduous task as detailed in the previous section (§3.3.1), thus we explore the effect of varying the number of samples used for finetuning the model. To perform this experiment, we allocated 20% of samples from each application as a “test set”, and remaining dataset for training the finetuned models. The number of finetuning samples were to be 20%, 40%, 60%

and 80% of the overall sample, and the models were tested on the "test set" to obtain the metrics.

Figure 3.6 contains the analysis of model performance across varying finetuning sample sizes reveals a consistent pattern of improvement followed by plateau across all three models - Gemma, Llama 3.1, and GPT-4o. For metric queries, GPT-4o demonstrates superior performance, with accuracy increasing from 0.23-0.37 at 20% samples to 0.74-0.79 at 80% samples across applications. Similarly for log queries, GPT-4o achieves F1-scores ranging from 0.26-0.38 with 20% samples, improving to 0.66-0.76 with 80% samples. The performance gains are most pronounced when increasing from 20% to 60% of the training data, after which the improvements become marginal. For instance, GPT-4o's accuracy on HDFS metrics increases substantially from 0.28 (20%) to 0.78 (60%) but only marginally to 0.79 (80%). This plateau effect is consistent across models and applications, suggesting that around 60% of the training data captures most of the log formats and lines that are present in the logs for a particular application.

Looking at individual models, Gemma shows the most modest improvements, with metric accuracy increasing from 0.12-0.18 at 20% to 0.39-0.55 at 80% across applications. Its F1-scores follow a similar trend, rising from 0.18-0.24 to 0.37-0.47. Llama 3.1 demonstrates slightly better scaling, achieving accuracies of 0.52-0.57 and F1-scores of 0.45-0.58 at 80% samples, up from 0.10-0.14 and 0.10-0.17 respectively at 20%. GPT-4o consistently outperforms both models across all sample sizes and applications. The HDFS application generally sees better performance compared to OSTK and OSSH across all models, particularly in the higher sample percentage ranges. The consistent plateauing behavior across all models and applications suggests an inherent limit to how much performance can be improved simply by increasing the amount of finetuning data. The difference in the results as more samples can be attributed to the LLMs understanding more information from the logs, which

was a common problem in models pre-finetuning. Since each LLM learns the log patterns differently, there is a need for understanding the amount of samples required to finetune the LLMs, and future studies need to be conducted on understanding the interactions between the LLM, log query language and the amount of data required to finetune these models.

Increasing the number of finetuning samples generally improves model performance in generating LogQL queries, with most gains achieved by 60% of the training data, after which returns diminish significantly. While GPT-4o consistently outperforms Llama 3.1 and Gemma across all sample sizes, all models exhibit similar plateauing behavior, suggesting an inherent limit to performance improvements through increased training data alone.

3.4.3 Transferability of the finetuned models

To evaluate the models' generalization capabilities across different applications, we conducted cross-application experiments where models were finetuned on two applications and subsequently tested on a third, previously unseen application. For instance, to assess query generation capabilities for OSSH logs, the models underwent finetuning using query datasets from OSTK and HDFS applications, thereby testing their ability to transfer learned patterns to a novel application context.

Model Name	OSSH		OSTK		HDFS	
	Metric	Log	Metric	Log	Metric	Log
Llama	0.13	0.14	0.11	0.07	0.05	0.1
Gemma	0.09	0.07	0.12	0.18	0.09	0.16
GPT-4o	**0.32**	**0.33**	**0.22**	**0.47**	**0.27**	**0.29**

Table 3.4: Results for transferability of finetuned models across applications.

Table 3.4 presents the performance metrics of models after finetuning on two applications and evaluating on a third application. The results reveal that while cross-

application fine-tuned models are inferior to application-specific fine-tuned models in most cases, they generally outperform their non-finetuned counterparts.

For example, GPT-4o demonstrates strong performance across all applications, achieving metric accuracy between 0.22 and 0.32 and log query F1-scores ranging from 0.29 to 0.47. Importantly, fine-tuning GPT-4o on OSTK and HDFC leads to significant relative improvements as compared to the non-fine-tuned counterparts, with performance gains of 52% (from 0.21 to 0.31) and 106% (from 0.16 to 0.33) in OSSH metric accuracy and log queries, respectively.

Moreover, smaller models like Llama and Gemma show minor improvements with respect to their non-fine-tuned versions. Llama's metric accuracy ranges from 0.13 on OSSH to 0.05 on HDFS, while Gemma shows inconsistent metric accuracy, ranging between 0.09 and 0.12. These marginal improvements in accuracy and F1-scores can be attributed to the models ability to capture syntactic patterns from the fine-tuning dataset.

However, the overall limited performance of these models stems from insufficient exposure to application-specific log patterns and their corresponding log query labels during training. Consequently, the errors observed in these models mirror those of their non-finetuned versions, particularly in their inability to effectively incorporate application-specific log information.

Evaluation of cross-application transferability revealed that models finetuned on two applications demonstrate limited performance when tested on a third, unseen application. Despite GPT-4o showing relatively better performance with metric accuracy up to 0.32, all models exhibited performance levels closer to their non-finetuned versions, primarily due to insufficient exposure to application-specific log patterns and corresponding query labels.

Model	OSSH	OSTK	HDFS
Llama	0.57	0.58	0.59
Gemma	0.43	0.39	0.46
GPT-4o	**0.78**	**0.86**	**0.77**

Table 3.5: Codebert score for various models and applications

3.4.4 Code Quality Analysis

For analyzing the quality of the code produced by various finetuned models, we make use of CodeBertScore[154] (CBS) to evaluate the quality of the LogQL query generated by the finetuned models.

Finetuning CBS Since the current CBS model doesn't support LogQL, we finetuned the CBS model to be able to score the outputs for the model. We used 50% of application specific LogQL queries to finetune the CBS model, and used 30% of the dataset to finetune the LLMs, and tested it on 20% of the dataset by comparing the output of the finetuned models with the correctly written LogQL queries in the dataset. Table 3.5 shows the CBS for the queries generated by the finetuned models. GPT-4o consistently achieves the highest CodeBERTScore across all applications, with scores of 0.78 for OpenSSH (OSSH), 0.86 for OpenStack (OSTK), and 0.77 for HDFS. These high scores indicate that GPT-4o generates LogQL queries that are highly similar to the reference queries, both semantically and functionally. In contrast, the Gemma model exhibits significantly lower scores, ranging from 0.39 to 0.46, suggesting that its generated queries are less aligned with the reference queries and may require substantial refinement to achieve functional correctness. The Llama model performs moderately, with scores between 0.57 and 0.59, indicating partial similarity but room for improvement in terms of query accuracy and efficiency. Overall, these results highlight the importance of selecting models with higher CodeBERTScores to ensure the generation of high-quality LogQL queries that are both syntactically correct and functionally reliable.

The evaluation demonstrates that finetuned LLM models can successfully generate LogQL queries, with varying degrees of accuracy across different models. GPT-4o emerged as the top performer, whereas other models showed moderate to lower performance, with Llama achieving scores between 0.57-0.59 and Gemma scoring between 0.39-0.46, suggesting their generated queries require more refinement to achieve full functional reliability.

3.5 Discussion

This work showed that there is a necessity for enhanced interfaces in observability data query generation. To our knowledge, this research presents the first comprehensive data collection effort for fine-tuning models to generate log query language. Our evaluations demonstrate that while base LLMs exhibit limitations in generating LogQL queries, fine-tuning these models significantly enhances their query generation capabilities. Although we present a proof of concept across various applications, practitioners seeking to implement our methodology would need to develop a corpus of natural language to LogQL queries specific to their applications, as accurate query generation necessitates understanding application-specific log semantics. Additionally, the model must generate syntactically valid queries compatible with their internal system's query language. These considerations directly influence the selection of base models for fine-tuning purposes. For instance, organizations utilizing Datadog for observability data storage would need to develop a dataset mapping Natural Language to DQL queries, and select base models capable of DQL query generation. This dataset must encompass diverse query types addressing the three domains outlined in Section 3.3.1. This research aims to serve as a framework for organizations developing observability data queries within their specific contexts.

3.5.1 Threats to Validity

The fine-tuned LogQL-LM models demonstrate superior performance metrics across both metric and log queries compared to the baseline model. However, as this research represents one of the initial endeavors in LogQL query generation, several limitations, and opportunities for improvement warrant discussion.

A significant limitation observed in the current implementation is the models' reduced efficacy in generating LogQL queries when there exists semantic divergence between the natural language (NL) query and the corresponding log entries. To illustrate, consider the example in Table 3.1-row 3, where the objective is to identify recent successful authentication events for user "fztu". While the NL query employs the phrase "successful login," the actual log entries utilize "accepted password" to denote such events. Consequently, the model frequently generates queries containing "successful login," resulting in null result sets. This limitation can be attributed to two primary factors. First, the methodology of reverse-engineering natural language (NL) questions from existing dashboards inherently limits the diversity of NL queries that can map to a specific LogQL query. Contemporary Text-to-SQL benchmarks attempt to address this limitation by manually writing multiple natural language questions for a SQL query, which can easily get cumbersome. One potential ancillary direction is to design automated paraphrasing techniques to generate semantically equivalent and useful NL queries. Second, current log aggregation systems, including Loki, lack semantic search capabilities within log data. The implementation of semantic similarity matching in search systems would enable LogQL queries containing phrases like "successful login" to successfully retrieve log entries containing semantically equivalent terms such as "accepted password."

An interesting direction for future research is exploring how to achieve additional performance gains in generating viable LogQL queries for application logs absent from the fine-tuning dataset. As demonstrated in Section 3.4.3, while fine-tuning

successfully addresses certain syntactical challenges, enhancing model transferability necessitates the incorporation of diverse application logs during the fine-tuning process. This diversity spans across logging frameworks, application domains, and implementation patterns, creating substantial complexity in sampling representative log entries. Although utilizing application logs for fine-tuning could reduce dependence on manually created NL-to-LogQL pairs, the challenge of sampling a sufficiently diverse and representative set of logs remains an open research problem [94].

Our work focuses on generating LogQL queries directly from natural language, presenting an alternative to the widely used practice of constructing LogQL queries through Grafana dashboards. The existing workflow, which requires users to migrate queries between our interface and Grafana, introduces significant cognitive overhead due to constant context switching and hampers developer productivity. To address this, we envision an intelligent interface, akin to GitHub Copilot, that provides contextual suggestions for query entities, labels, and log lines during composition. However, realizing this vision involves several technical challenges: (a) the need for extensive training data mapping natural language to LogQL queries, (b) meeting stringent latency requirements for real-time autocompletion, and (c) supporting bidirectional context processing, which diverges from the conventional left-to-right generation approach in NL-to-LogQL systems. Overcoming these challenges would require restructuring our dataset to focus on LogQL queries and adopting a Fill in the Middle (FIM) architecture to facilitate the predictive completion of query prefixes and suffixes during interactive development.

```
How many times did we receive a 503 status
code while validating tokens in the past
30 days for openstack-asia-pacific?
```

(a) Summarized NL Query, complete NL query in our code repository

```
count_over_time({job="openstack", region
="asia-pacific"} |= "503" |= "token
validation" [30d])

calculate_over_time({job="openstack",
region="asia-pacific"} |= "503" !=
"token" [30d])
```

(b) LogQL Query (Green: Correct, Red: LLM Generated)

```
@http.status_code:503 service:openstack
-asia-pacific "validate" @timestamp:>now-30d

status.code:503 service_name:openstack-asia-pacific
   ↪ "validate" @timestamp:>now-30d
```

(c) Datadog Query (Green: Correct, Red: LLM Generated)

```
grep -E "status:␣503.*validate_token" OpenStack_2k.log |
   ↪ grep "openstack-asia-pacific" | wc -l

grep "ERROR␣503" OpenStack.log | awk '{print␣$4}' | sort |
   ↪ uniq -c
```

(d) `grep`-Based Query (Green: Correct, Red: LLM Generated)

Figure 3.2: Queries for analyzing 503 status codes in OpenStack Asia-Pacific across different query languages

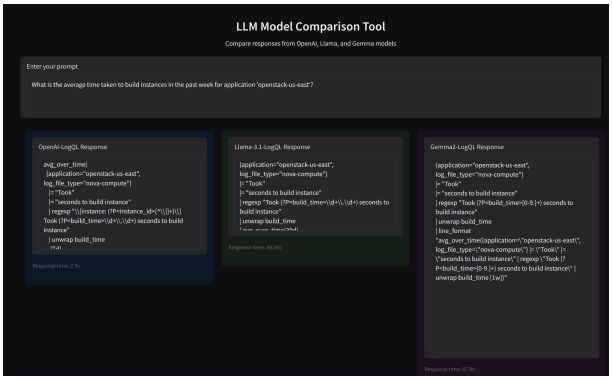


Figure 3.3: Demonstration of the model.

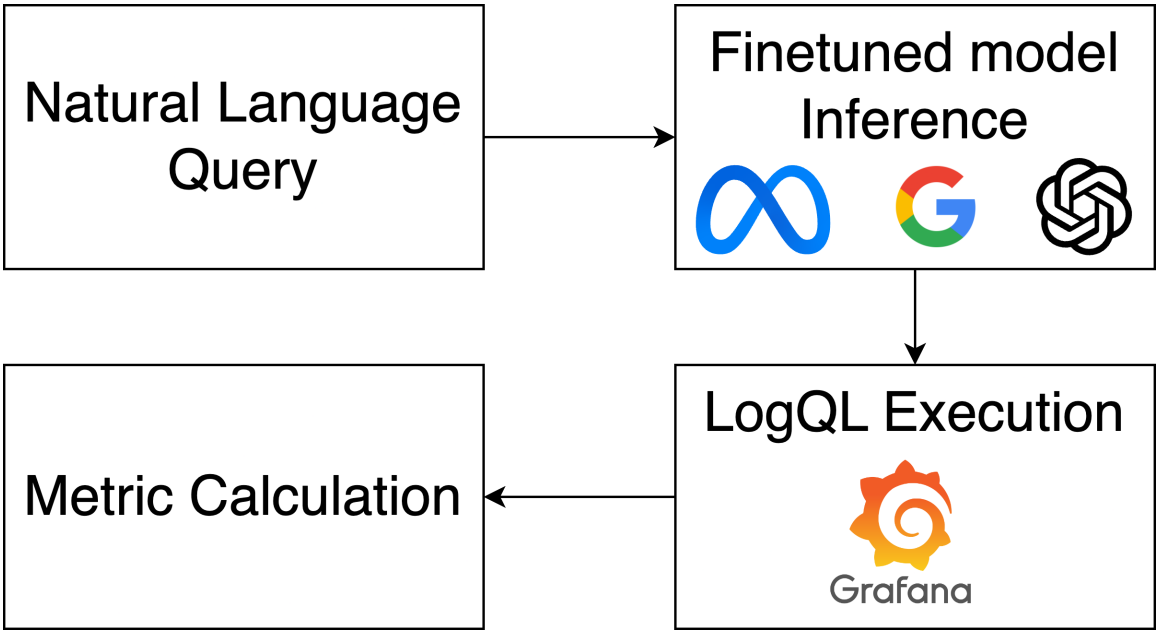


Figure 3.4: Evaluation Pipeline

```
topk(1, sum by (source_ip)(count_over_time(
{application="hdfs-us-east", component=
"dfs.DataNode$DataTransfer"} \|~ "Transmitted
block␣.*␣to␣.*" \| regexp "(?P<source_ip>[\\d\\.]+)
:\\d+:Transmitted␣block␣.*␣to␣.*" [12h])))

topk(1, sum (source_ip) ( count_over_time(
{app="hdfs-us-east", component=
"dfs.DataNode$DataTransfer"} |~ /*[12h]*/
"Transmitted␣block␣.*␣to␣.*"  | ))
```

(a) LogQL query for NL Query to find the node with most number of successfull block transmissions. The wrong query contains time aggregration in the middle of the query. [GPT-4o]

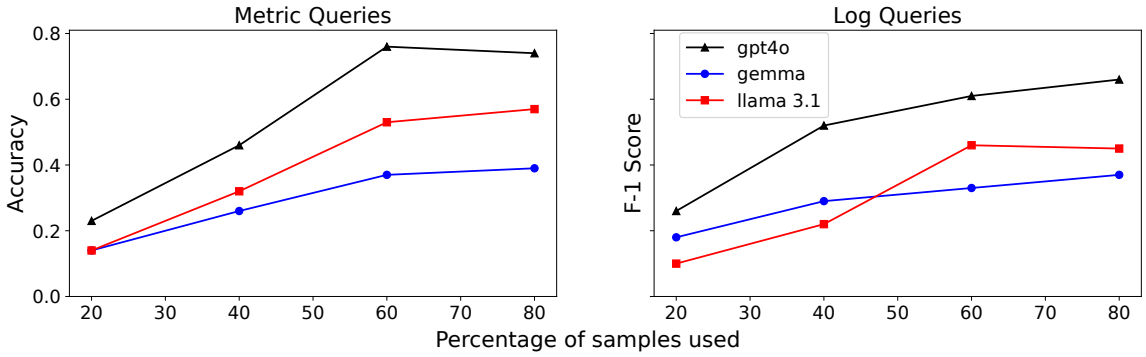
```
{application="openssh"} |= "Did␣not␣receive
identification␣string␣from" | hostname="LabSZ
-tenant-5" | line_format "‘{{__timestamp__}}
‘-␣Failed␣to␣receive␣identification␣string
from␣{{.content}}"

{hostname!="LabSZ-tenant-5", /*app ="ssh"*/}
!= "Did␣not␣receive␣identification␣string
from" | line_format "{{/*timestamp*/}}␣-␣No
identification␣from␣{{.message}}"/*[1m]*/
```

(b) List of all instances where there was a failure in receiving an identification string from host 'LabSZ-tenant-5'. The wrong query contains wrong label (app), wrong timestamp format and improper time aggregation. [Llama]

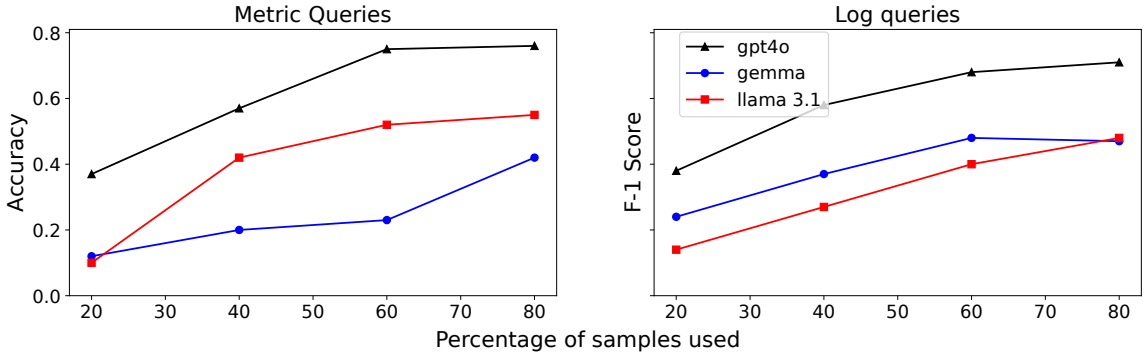
```
sum by (component) ( count_over_time(
{application="openstack-eu-west", component
="nova.virt.libvirt.imagecache"}|~ "Active
base␣files:␣(?P<file_path>/.*)"[1h]))

sum by (component) ( count_over_time /*by*/
(file_path)({application="openstack-eu-west",
component="nova.virt.libvirt.imagecache"}
|~ "Active␣base␣files:␣(?P<file_path>/.*)") )
```

(c) LogQL query for retrieving the total size of all base files in openstack-eu-west. This query lacks time aggregation ([1h]) block, wrong syntax of using "by" which is present in the correct query. [Gemma]

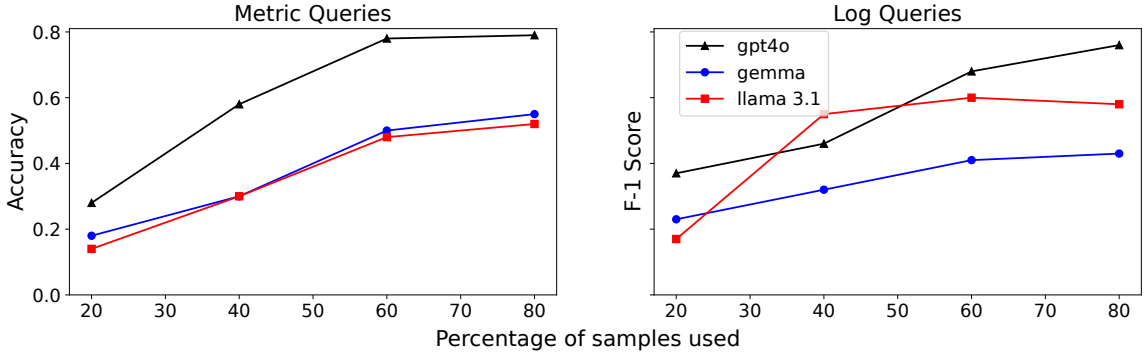
Figure 3.5: Examples of logql queries generated by the models before (black color, with errors in red) and after (green color) finetuning.



(a) OSTK



(b) OSSH



(c) HDFS

Figure 3.6: Model accuracy with different number of samples in finetuning phase

Chapter 4

Sauron: Semantic Search Engine

4.1 Introduction

Logs are a cornerstone of system observability, providing critical insights into the runtime behavior of software systems. However, traditional methods of querying and analyzing logs face significant challenges, especially in the context of modern microservice architectures. As observed in the previous chapter, the NL2LogQL model struggled to generate accurate queries when natural language (NL) questions did not contain specific keywords used in log query languages. This limitation highlights a broader issue: the disconnect between those producing logs (developers) and those analyzing them (e.g., Site Reliability Engineers, or SREs). Developers often design logs based on their own understanding of the system, while SREs or other engineers tasked with troubleshooting may lack familiarity with the terminology and structure of these logs. The complexity of modern software systems is further compounded by the growing adoption of microservices architecture, where developers frequently work on isolated components and may not fully understand the vocabulary embedded in logs generated by other teams or services. This architectural paradigm, while offering benefits in terms of scalability and deployment flexibility, introduces significant challenges in log

comprehension and analysis during debugging scenarios.

Considering a typical microservice architecture shown in Figure 2.1, each service is owned by a different team and other teams access the service using a typical request-response model. This organizational structure creates natural boundaries between development units, with each team developing expertise in their specific domain while potentially remaining unfamiliar with the internal workings of services maintained by other teams. The communication between these services occurs primarily through well-defined APIs, with the implementation details of each service remaining largely opaque to consumers.

When newer versions of the downstream service are deployed, a subsequent change on the upstream service has to be made to accommodate these modifications. These changes may involve adapting to new API signatures, handling different response formats, or adjusting to altered behavior patterns. Due to the nature of modular teams in software organizations, these changes may not be implemented by the same developer who deployed the previous iteration of the upstream service. This discontinuity in developer involvement creates a knowledge gap, where the contextual understanding that informed the original implementation is not fully transferred to subsequent maintainers.

While changing the codebase, developers typically also rewrite the log lines that are present in the codebase, and might not use the original words that were in the previous log line. This divergence in logging vocabulary occurs for various reasons: different developers have distinct writing styles, terminology preferences, and mental models of the system. Additionally, as requirements evolve and implementations change, the logging needs and focus areas shift accordingly. This leads to a situation where the current log text typically contains similar information to the previous log text, while being expressed using different terminologies.

The terminologies are often system-specific; for example, in the case of HDFS logs,

the various blocks in the log files could be indicated as “block_id” or “blk_id”. Such variations, while seemingly minor, create significant challenges when attempting to search across logs to identify patterns or troubleshoot issues. Using shorthand versions of variable names is a common strategy for reducing the size of the log files, which can often grow into hundreds of terabytes of data. While this practice is justified from a storage efficiency perspective, it further exacerbates the problem of log comprehension, especially for developers who did not author the original code. The heterogeneity in logging practices extends beyond variable naming to include message structures, verbosity levels, and contextual information inclusion. Some developers might prefer concise logs that capture only essential information, while others might opt for more verbose logs that provide comprehensive context. This lack of standardization makes it challenging to develop universal tools for log analysis and search.

A recent study at Microsoft[65] revealed that engineers often analyze logs they are unfamiliar with, leading to significant delays in diagnosing failures. Approximately 42% of surveyed engineers reported spending over an hour analyzing logs during incidents, underscoring the inefficiency of existing processes. This statistic highlights the real-world impact of the log comprehension problem, translating directly into extended downtime, delayed issue resolution, and increased operational costs. Moreover, correlating logs from various sources remains a challenge due to inconsistent terminology and heterogeneous formats, further complicating the troubleshooting process in distributed systems.

When searching within observability data, specifically logs or traces, it is common to rely on “wildcard queries” to perform approximate searches in order to retrieve relevant log lines [77]. These queries typically involve pattern matching with wildcards (e.g., “*password*”) to capture variations in phrasing. However, this approach is fundamentally limited by its reliance on lexical matching rather than semantic understanding. The lack of standardized language in these files, often application or

context-specific, poses challenges for standardization efforts that require substantial resources for enforcement. Even with organizational guidelines in place, achieving perfect consistency in logging practices across large development teams remains an elusive goal.

Existing products primarily offer full-text search capabilities [76], where users must have prior knowledge of at least a portion of the stored text, limiting their ability to search files based on contextual meaning. These systems typically employ inverted indices that map terms to documents, enabling efficient retrieval based on keyword matches. However, they fail to capture the semantic relationships between terms or the contextual meaning of phrases. For instance, if a developer wishes to log successful password authentication, they might use phrases such as "Authenticated Password" or "Password Accepted", making it difficult to query the data without knowing the exact phrase. A search for "successful login" would fail to retrieve these logs despite their semantic relevance to the query.

Most large-scale applications are created by multiple developers, and not all of them can be expected to remember all the specific phrases or text conventions within these files. As systems grow in complexity and team sizes increase, the cognitive load of maintaining awareness of all logging conventions becomes unsustainable. This predicament highlights the need for semantic search on observability data. Semantic search goes beyond keyword matching by understanding the contextual meaning of search queries and documents, harnessing natural language processing to analyze the semantics of the query and the underlying data, thus providing more precise and relevant search results based on the intended *meaning* rather than just the literal *matching* of terms.

Currently, observability data is used by developers for various tasks like capacity planning, debugging and other activities required to build and maintain these global scale systems. Yet the potential insights from the collected log data can have high

utility in other business-related decisions. For instance, the insights from HTTP header parsing can be used for understanding the regions from where the site is receiving traffic to optimize marketing campaigns, or the distributed traces can be analyzed to get the most commonly used features of a platform. By asking questions like "*what is the average response time for my website?*" or "*what are the most common errors being reported?*" organizations can paint a more detailed picture of how their systems are performing and identify areas that need improvement.

Log analytics serves as a powerful tool for business intelligence, enabling data-driven decision-making that can provide competitive advantages and drive business growth. Through analyzing user behavior, transaction logs, and application usage patterns, organizations can gain deeper understanding of customer preferences and market trends. For example, e-commerce companies can leverage logs to track product popularity, customer interactions, payment trends, and inventory levelstransforming raw data into actionable intelligence for strategic decisions. These insights can help businesses optimize marketing strategies, improve user experiences, and personalize recommendations based on actual customer behavior rather than assumptions.

Furthermore, log data can reveal operational efficiencies and inefficiencies, helping businesses streamline processes and reduce costs. Historical log analysis enables organizations to identify trends and patterns that might indicate potential future issues, allowing for proactive rather than reactive approaches to business challenges. However, if a non-engineer wants to extract information from this data, they need to rely on an engineer to write queries, or learn the querying language themselves. This limitation highlights the need for semantic search capabilities that would democratize access to valuable log insights across the organization, allowing business leaders to make informed decisions without technical barriers.

Given these challenges, there is a pressing need for a "semantic search system" tailored for log data. This system would enable intuitive and efficient querying by

Dec 12 23:54:44 reverse mapping checking getaddrinfo for 191-210-223-172.user.vivozap.com.br [191.210.223.172] failed - POSSIBLE BREAK-IN ATTEMPT!

(a) Example log line from OpenSSH Server Logs [30]

```
> cat sshlogs.log | grep \BREAK-IN" | uniq
```

(b) Example `grep` command for searching the break in attempts.

List all the IP addresses trying to hack?

(c) NL Query

51.15.196.223, 191.210.223.172, 51.15.203.45.

(d) Response from Sauron

Figure 4.1: Example use case for both traditional querying method, and Sauron.

understanding the intent behind user queries rather than relying solely on keyword matching. This approach could significantly reduce the time required to locate relevant information, streamline cross-team collaboration, and improve failure diagnosis. By bridging the gap between log producers and consumers, semantic search systems have the potential to transform log analysis workflows in modern software environments.

4.2 Sauron

Inspired by the problems in 4.1, and leveraging the recent advances in Natural Language Processing and Information Retrieval, the main issue with log search systems stems from their reliance on inverted indexes as the fundamental search mechanism. Inverted indexes map words to the documents that contain them, creating an efficient data structure for retrieving exact or very similar variants of given words. While highly effective for lexical matching, this approach significantly constrains user queries to contain the exact text from log linesan unrealistic expectation given both the massive scale of generated logs and the number of developers working on these systems.

Traditional inverted indexing operates by processing documents to create a list of all unique terms and their corresponding document identifiers. When a query is submitted, the search engine looks up these terms in the inverted index to quickly identify and retrieve relevant documents. This process, while computationally efficient, fundamentally limits search capabilities to lexical matching rather than semantic understanding. For instance, when a user searches for "accepted," an inverted index will only identify documents containing that exact word, missing semantically equivalent terms like "authenticated", which relies on the contextual information.

In contrast, embedding models transform text into dense vector representations that capture semantic meaning and contextual relationships. These embeddings position semantically similar items close to each other in high-dimensional vector spaces, enabling more nuanced and accurate processing of textual data. Mathematically, contextual embeddings can be represented as a function that maps a sequence of words, denoted as $x = [x_1, x_2, \ldots, x_n]$, to their corresponding embeddings, denoted as $h = [h_1, h_2, \ldots, h_d] \in \mathbb{R}^d$ where d is the dimensionality of the embedding vector. The key characteristic of contextual embeddings is their sensitivity to the context of input words – the embedding of a word depends not only on the word itself but also on its surrounding words. These models employ self-attention mechanisms to capture dependencies between words in the input sequence, effectively encoding contextual information and generating meaningful embeddings. This semantic capability makes embedding models particularly valuable for log search systems, where technical terminology and system-specific language often require contextual understanding beyond simple keyword matching.

However, off-the-shelf embedding models are not always optimized for specific domains like log data[115]. This is where fine-tuning becomes crucial. Fine-tuning an embedding model on domain-specific data enhances its ability to capture contextual nuances particular to log formats and technical terminology. The process involves

starting with a pre-trained model that already performs well on general tasks, then re-training it on domain-specific datasets. This adjustment allows the model to better reflect the unique vocabulary and semantics of log data, significantly improving search relevance and accuracy.

4.2.1 System

SAURON employs a sophisticated dual-phase architecture designed for efficient semantic analysis of observability data. The system architecture is methodically bifurcated into the Indexing phase and the Querying phase, each serving distinct yet complementary functions within the overall workflow.

The Indexing phase details the data processing pipeline by ingesting log files through multiple input vectors – either via direct file upload mechanisms or through integration with distributed streaming platforms such as Apache Kafka. Upon ingestion, the system applies advanced natural language processing techniques to generate high-dimensional vector embeddings that capture the semantic essence of the textual data. These embeddings are subsequently persisted in a specialized vector database optimized for high-throughput storage and retrieval operations, creating a comprehensive searchable index of the observability corpus.

The querying phase consists of a modular RAG framework[59], where a user submits a query, the system first splits the user query into metadata information and the actual query text. The query text is transformed into the same embedding space as the indexed documents. This query embedding is then utilized to perform approximate nearest neighbor (ANN) search operations against the vector database, employing algorithms that balance search accuracy with computational efficiency. The ANN search identifies semantically relevant documents based on vector similarity metrics, effectively retrieving contextually appropriate information regardless of exact keyword matching.

The retrieved documents, representing the most semantically relevant context for the query, are subsequently passed to a Large Language Model (LLM) alongside the original query. The LLM leverages this enriched context to generate comprehensive, contextually-aware responses that address the user's information needs with high precision and relevance, effectively bridging the gap between raw observability data and actionable insights.

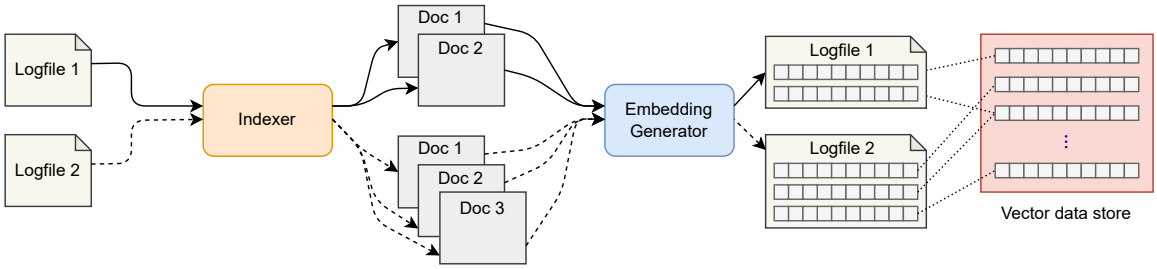


Figure 4.2: Indexing step architecture.

4.2.2 Indexing step

Figure 4.2 illustrates the indexing step in our semantic search system for log data. The process begins with the aggregation of data from various sources, followed by chunkingsplitting the data into smaller documents. This chunking strategy is crucial for maintaining appropriate granularity in the embeddings and ensuring optimal context size for the LLM. The diagram shows how these chunks then undergo embedding generation, where each document is transformed into a high-dimensional vector representation that captures its semantic meaning. These embeddings are subsequently stored in a specialized vector database optimized for Approximate Nearest Neighbor (ANN) search. Vector databases like Weaviate and Pinecone are purpose-built for this task, offering efficient storage and retrieval mechanisms specifically designed for high-dimensional vector data, enabling rapid similarity-based retrieval that transcends simple keyword matching. The ingestion pipeline depicted in the diagram demonstrates how raw log data flows through these sequential processing stages, each

contributing to the transformation of unstructured text into structured, searchable vector representations. This transformation is fundamental to enabling semantic search capabilities that understand the meaning behind queries rather than merely matching keywords. The embedding models employed in this process are trained on vast corpora of text, allowing them to capture complex semantic relationships and contextual nuances within the log data, which is particularly valuable when dealing with the technical and often domain-specific language found in system logs.

While ANN search provides efficient retrieval, it's important to acknowledge that the chunk with the highest similarity score may not always be the most relevant to a user's query. To address this limitation, our system retrieves the top-K most similar chunks (typically K=50 or 100) and combines them to form a comprehensive context. The diagram illustrates this multi-retrieval approach, highlighting how multiple chunks are combined to provide richer context for subsequent processing. Empirical benchmarks have demonstrated that increasing the value of K improves recall scores, approaching 1 as K grows, indicating that the truly relevant information is almost always present within the retrieved set, even if not ranked first[32]. This carefully designed ingestion pipeline creates the foundation for our semantic search system, enabling efficient and meaningful exploration of log data beyond the capabilities of traditional keyword-based approaches, while ensuring that the most relevant information is included in the context provided to the LLM. The vector database's indexing structures, as shown in the diagram, are optimized to handle the high-dimensional nature of embeddings, employing hierarchical navigable small worlds (HNSW) to partition the vector space and enable logarithmic-time search complexity instead of linear scanning. This optimization is critical when dealing with large volumes of log data, where response time requirements may be stringent. Additionally, the system supports incremental updates to the vector database, allowing new log entries to be processed and made searchable without requiring a complete re-indexing of the

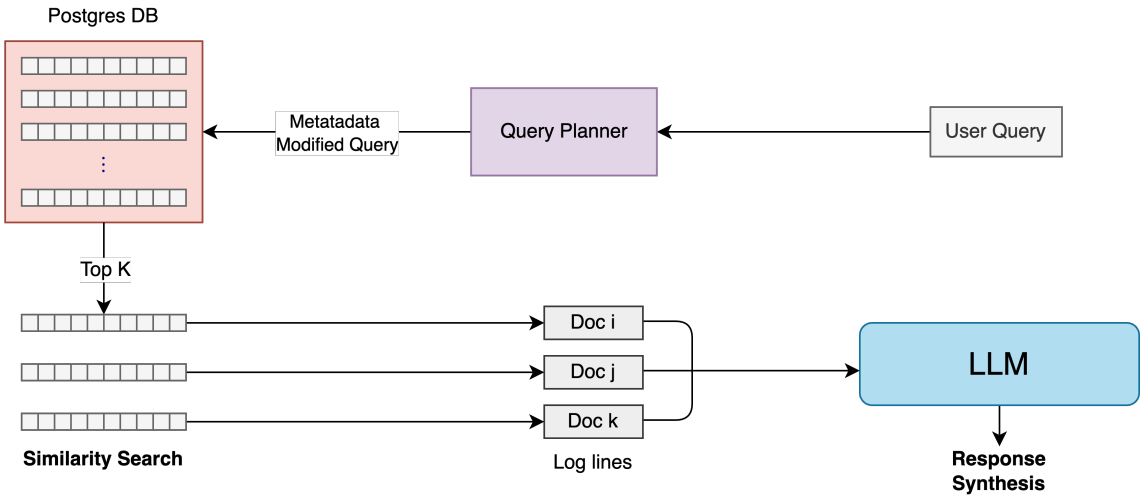


Figure 4.3: Querying Step.

entire dataset, thus maintaining the freshness of searchable content while minimizing computational overhead.

4.2.3 Querying Step

Querying step consists of a modular RAG framework[59] where the sophisticated processing pipeline enables efficient extraction of valuable insights from massive log datasets distributed across complex systems. In the querying step, the user's natural language (NL) query first goes to a query planner module, that is used as a pre-retrieval step to identify the application(s) or services required to run a particular query, the various indexes, timestamps and other application specific metadata. This critical pre-processing phase significantly reduces the computational burden by narrowing the search space before initiating resource-intensive vector similarity operations across terabytes of log data generated daily in enterprise environments. Along with this metadata, the model rewrites the incoming user query such that it aligns more with the application specific context from the logs. This query rewriting process incorporates domain-specific knowledge about log formats and system architecture, enabling more accurate information retrieval by transforming ambiguous natural language

into structured queries that better match the underlying data representations. For eg, if the incoming user query is "List all the blocks that started between 10 AM to 12 PM on node 1 yesterday", the information about the start and end times, and the specific node is metadata whereas the actual user query is requesting the blocks that started during that time. In this query for the HDFS logs, the query planner recognizes temporal constraints, node identification, and the entity of interest (blocks), separating these components for specialized processing while preserving the fundamental information need. The query planner layer separates the metadata and rewrites the query for internal representation, transforming natural language into a hybrid query format that leverages both traditional SQL capabilities for metadata filtering and vector-based semantic matching for content relevance. This separation enables a dual-path processing architecture where computationally efficient metadata filtering occurs before more resource-intensive semantic similarity operations, dramatically improving response times compared to naive approaches. Sauron uses postgres database for storing both the raw logs, and also the pgvector extension for storing the embeddings, leveraging the enterprise-grade reliability of PostgreSQL while extending its capabilities with specialized vector search functionality essential for semantic matching of unstructured log content. In the log files, these blocks could be represented using "blk_id" which is captured by the embeddings that is stored in the vector store, allowing for contextual understanding of technical identifiers that would be opaque to traditional keyword-based search methods. These embeddings are generated through specialized encoding models that transform raw log text into high-dimensional vectors that preserve semantic relationships while accommodating the unique characteristics of system logs, including their terse syntax, domain-specific abbreviations, and structured format. The vector store is partitioned based on application, and each embedding has the metadata information about the node from which it was collected, and also the timestamp as well, enabling efficient multi-dimensional

filtering before executing computationally expensive similarity searches. This partitioning strategy reflects real-world system architectures where logs from diverse applications exhibit significantly different formats, vocabulary, and semantic structures, necessitating specialized processing pipelines for optimal retrieval performance. The modified user query's metadata component is used to filter the embeddings which fall within the time range and are from the relevant node, leveraging PostgreSQL's highly optimized indexing mechanisms to rapidly eliminate vast portions of the log corpus from consideration. In production environments processing petabytes of log data, these metadata filters routinely reduce the candidate set by 99.9% or more before vector similarity calculations begin, transforming potentially hour-long searches into operations completing in seconds.

As shown in previous chapter (§3.2.2), these labels when stored using loki are more deterministic and can be queried easily, whereas the actual log lines are often unindexed, presenting significant challenges for traditional text-based search approaches that rely on exact pattern matching. The integration with Grafana Loki enhances our system with established log aggregation capabilities while our vector-based approach addresses Loki's limitations in semantic understanding of log content. Once the relevant embeddings are filtered using the metadata, similarity search is run using the embeddings of the modified query to identify the top-k relevant log lines, implementing approximate nearest neighbor algorithms optimized for high-dimensional vector spaces. These algorithms balance search efficiency against result quality through sophisticated index structures that avoid exhaustive comparison while maintaining high recall rates, critical for production deployments where both performance and accuracy are non-negotiable requirements. In this case, we use a large value for k to increase the recall score closer to 1, acknowledging the practical reality that downstream processing can refine results but cannot recover relevant items missed during initial retrieval. The empirically determined optimal value for k varies based on log characteristics and

query complexity, typically ranging from several hundred to a few thousand candidates in production deployments processing billions of log entries. These top-k log lines and the original query are passed to the LLM as context, and the LLM further filters out the log lines that are not relevant, while also re-ranking the logs based on relevance metrics, effectively performing a second-stage retrieval refinement that leverages the deeper semantic understanding capabilities of large language models. This two-stage retrieval architecture combines the computational efficiency of traditional information retrieval with the sophisticated reasoning capabilities of foundation models, addressing fundamental limitations inherent to either approach used in isolation. It also provides a concise summary of the actual entity that was requested, transforming raw technical data into human-readable insights through abstractive summarization techniques that preserve critical information while eliminating extraneous details. As shown in Figure 4.1, it provides the IP addresses as output along with the relevant log lines, demonstrating the system's practical utility in security operations where rapid identification of suspicious network activity across massive log volumes can mean the difference between preventing an intrusion and detecting a breach after significant damage has occurred.

4.3 Evaluation

In §4.2, we proposed the system, and implemented it using Python. For the logs, we made use of the logs from the previous chapter, and used the log templates from LogHub dataset[158]. The semantically related questions were human generated using the natural language questions from the previous chapter. Since the existing embedding models do not have context on log specific vocabulary, a lot of which is not traditional english language, we finetuned existing base model "multi-qa-mpnet-base-cos-v1" to generate embeddings. Using the finetuned embedding model, and the

Embedding model	Application	NDCG@10	NDCG@50	NDCG@100
Base Model	HDFS	0.12	0.32	0.87
	OSTK	0.18	0.31	0.85
	OSSH	0.15	0.35	0.84
Finetuned Model	HDFS	0.26	0.64	0.91
	OSTK	0.28	0.63	0.92
	OSSH	0.22	0.61	0.91

Table 4.1: Performance of the trained embedding model compare to the base model

system described in §4.2, we answer the two research questions.

1. How effective is the finetuned embedding model compared to the base model?

2. How effective is Sauron compared to NL2LogQL?

4.3.1 Embedding Model Performance

The experimental results demonstrate the substantial benefits of fine-tuning embedding models for log search applications. Across three distinct datasets (HDFS, OSTK, and OSSH), the fine-tuned model consistently outperforms the base model by significant margins. Most notably, NDCG@10 scores improved by 116.7%, 55.6%, and 46.7% for HDFS, OSTK, and OSSH respectively, indicating dramatically better precision at the top of search results. Similarly, NDCG@50 scores doubled across all applications, with improvements ranging from 74.3% to 103.2%. While both models perform well at NDCG@100, the fine-tuned model still maintains a consistent advantage of approximately 5-8% improvement. These results clearly indicate that domain-specific fine-tuning creates embedding spaces that better capture the semantic relationships within log data. By adapting the vector representations to the unique vocabulary, syntax, and contextual patterns of system logs, fine-tuned models enable more accurate semantic search. The most significant improvements occur at lower NDCG thresholds (10 and 50), which are particularly important as they represent the most relevant results that users typically encounter first. This pattern suggests that fine-tuning

Application	Model	Log queries (Base)	Log Queries (Semantic)
HDFS	NL2LogQL	0.72	0.57
	SAURON	0.83	0.84
OSSH	NL2LogQL	0.64	0.59
	SAURON	0.89	0.86
OSTK	NL2LogQL	0.69	0.58
	SAURON	0.84	0.83

Table 4.2: End to end system performance of Sauron

particularly enhances precision for the most relevant matches, making it an essential technique for practical log search applications where users rarely examine results beyond the first few log lines.

4.3.2 End to End Log Search

Our experimental results demonstrate that Sauron, a modular RAG-based semantic search system, significantly outperforms NL2LogQL across all tested applications. In the case of HDFS, Sauron achieves F1 scores of 0.83 and 0.84 for base and semantic queries respectively, compared to NL2LogQL's 0.72 and 0.57. This performance gap is even more pronounced in OpenSSH, where Sauron reaches 0.89 and 0.86 F1 scores against NL2LogQL's 0.64 and 0.59. Similarly, for OpenStack, Sauron performs better with 0.84 and 0.83 F1 scores versus NL2LogQL's 0.69 and 0.58. Notably, while NL2LogQL experiences a substantial performance degradation when transitioning from base to semantic queries (averaging a 15.7% decrease), Sauron maintains consistent performance across both query types, with minimal variation (averaging only a 1% difference). This consistency reveals a fundamental limitation in NL2LogQL's approachdespite being fine-tuned on GPT-4O, it struggles with semantic variations that deviate from the exact log text patterns. In contrast, Sauron's RAG architecture effectively leverages vector embeddings to capture semantic similarities between query terms and log content, enabling it to recognize semantic differences such as "authentication failed" and "failed password attempt" refer to the same underlying

event.

The performance disparity is particularly significant in real-world observability scenarios where developers querying logs often use different terminology than those who wrote the log statements. Sauron's architecture addresses this vocabulary mismatch through its modular RAG design that separates embedding generation from retrieval mechanisms, allowing for specialized handling of domain-specific terminology. Furthermore, Sauron's consistent performance across both query types suggests it has developed a more robust internal representation of the relationship between natural language expressions and log semantics. These findings have important implications for observability systems in large-scale distributed environments like OpenSSH, where the ability to accurately retrieve relevant logs despite semantic variations can significantly reduce debugging time and improve system reliability. The modular nature of Sauron also provides greater flexibility for future enhancements, allowing individual components to be optimized independently as requirements evolve. These results confirm that Sauron provides a more reliable and adaptable solution for log analysis in complex distributed systems, especially when dealing with semantic variations in query formulation.

4.4 Discussion

In this chapter, we presented Sauron, a pioneering semantic search system specifically designed for log data analysis. This novel approach represents a significant advancement in the field by enabling direct natural language querying capabilities for log data, thereby eliminating the technical barriers traditionally associated with log interrogation systems.

Unlike existing approaches such as NL2LogQL, which necessitate an intermediary translation layer to convert natural language into structured query languages, Sauron

fundamentally reimagines the indexing paradigm for log data. By leveraging vector embeddings rather than conventional inverted indexes, our system establishes a semantic foundation that facilitates more intuitive and effective search operations within log repositories. This approach aligns with recent advances in semantic parsing for logs, as demonstrated by systems like SemParser[71], but extends beyond template extraction to enable comprehensive semantic understanding of log content.

The cornerstone of our system is a domain-specific embedding model fine-tuned to capture the syntactic peculiarities and semantic nuances inherent in diverse log formats. This specialized model demonstrates remarkable adaptability across heterogeneous logging systems, accommodating variations in structure, verbosity, and domain-specific terminology. Through rigorous evaluation, we have demonstrated that this embedding approach significantly outperforms traditional keyword-based search methods in retrieval metrics.

Sauron's modular RAG architecture provides a flexible framework that can be readily extended or customized to address specific organizational requirements. This architectural design ensures seamless integration with existing logging infrastructures while maintaining robust performance characteristics even when processing high-volume log streams. The decoupled nature of the retrieval and generation components allows for independent optimization of each subsystem, enhancing both efficiency and accuracy in log analysis tasks.

A particularly noteworthy contribution of Sauron is its democratization of log analysis capabilities. By providing an intuitive natural language interface, our system empowers not only specialized DevOps engineers but also developers across varying expertise levels to extract actionable insights from log data without requiring extensive knowledge of specialized query languages or log schemas[65]. This accessibility effectively reduces the cognitive overhead associated with troubleshooting and system analysis workflows. Our comprehensive evaluation experiments, conducted across three

different applications: HDFS, OpenStack and OpenSSH demonstrates Sauron's efficacy across multiple dimensions, including query processing latency, semantic matching accuracy, and user experience metrics. The results consistently indicate that Sauron achieves superior performance compared to existing solutions, particularly in scenarios involving complex query intentions or when analyzing logs with implicit semantic relationships.

While Sauron represents a significant advancement in semantic log analysis, we acknowledge several avenues for future research, including refinement of the embedding model to accommodate emerging log formats, integration of additional contextual information sources, and optimization strategies for ultra-large-scale log repositories. In conclusion, Sauron establishes a new paradigm for semantic log analysis that bridges the gap between human-centric query intentions and machine-interpretable log structures, ultimately transforming how organizations interact with and derive value from their log data.

Chapter 5

Conclusion

5.1 Conclusion

Recent changes in software development paradigms have sparked innovation across the computing stack. The advancement of communication mechanisms and deployment tools has accelerated the adoption of small modular services (microservices) that can be reused across organizational products. While this architectural shift enhances development velocity, it simultaneously decentralizes ownership and understanding of the software ecosystem. This decentralization, compounded by substantial developer turnover in large organizations, creates a knowledge gap where few developers possess comprehensive understanding of the systems they interact with daily.

To address this disparity, we first conducted a systematic investigation into industry-standard microservice implementation practices, examining developers' design considerations and comparing them with academic testbeds. This investigation revealed several *key insights*. Most significantly, we discovered that no existing benchmarks accurately represent the production services described by our study participants. Though unsurprising – given that each testbed was designed to investigate specific, narrowly defined questions – this limitation has inadvertently allowed researchers to

draw broad conclusions about increasingly complex systems without full acknowledgment of testbed constraints. While our work highlights these mismatches, we encourage researchers to consult the extensive microservice literature that addresses the individual topics we explore [141, 133, 83].

Our user studies revealed several unexpected characteristics of operational microservice systems. The presence of cycles in non-faulty production systems contradicted common assumptions and indicated that the topologies studied by the research community have been unnecessarily constrained. Additionally, we observed a striking lack of consensus among survey participants on fundamental questions such as "how would you describe microservices?", revealing confusion between microservices and shared libraries. This indicates a need for more precise characterization and definitions. Furthermore, hybrid and transitional monolith-microservice architectures were remarkably prevalent among our interview participants, further complicating established definitions and roles in this domain.

Based on these findings, we identified that observability tools require significant reconceptualization, as their foundational assumptions no longer align with microservice environments.

Our approach advances the state of the art by eliminating two critical prerequisites: expertise in LogQL query language and detailed understanding of log file structure and syntax. To facilitate this advancement, we developed a comprehensive dataset of 424 natural language queries, their corresponding LogQL implementations, and expected query results across extensive log files. Our methodology leverages fine-tuned large language models selected for their dual capabilities: understanding programming languages and query syntax to generate valid LogQL queries, and efficiently processing log files to interpret structural patterns. Empirical evaluation of our LogQL-LMdemonstrates substantial performance improvements post-fine-tuning, achieving success rates exceeding 75% in generating syntactically and semantically

correct LogQL queries comparable to those crafted by human experts.

While LogQL-LMaddresses LogQL query generation, advances in embedding technology suggest we might transcend intermediate query languages entirely. L everaging embeddings as fundamental building blocks enables powerful semantic search capabilities.

To this end, we developed Sauron, a RAG-based semantic search engine for log data that processes natural language queries directly.

By fine-tuning embedding models, we demonstrated the system's efficacy across three distinct applications. Notably, Sauron maintains consistent performance regardless of whether query text appears verbatim in log files, with only minimal performance degradation for novel queries.

This thesis contributes to the evolving landscape of distributed systems observability by building tools that address the real challenges faced by practitioners. By focusing on the needs of observers themselvesrather than assuming deep technical expertisewe hope to open new research avenues in log analysis and inspire the development of more accessible, powerful observability tools for increasingly complex distributed architectures.

Bibliography

[1] Apache Thrift Website. https://thrift.apache.org/.

[2] Bookinfo Application. https://istio.io/latest/docs/examples/bookinfo/.

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

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

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

[6] 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.

[7] Lessons From the Birth of Microservices at Google. https://dzone.com/articles/lessons-from-the-birth-of-microservices-at-google, .

[8] 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, .

[9] 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.

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

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

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

[13] Microsoft Microservice Evolution. https://www.slideshare.net/adriancockcroft/evolution-of-microservices-craft-conference, .

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

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

[17] Universal Description, Discovery and Integration (UDDI) Registry. https://access.redhat.com/documentation/en-us/jboss_enterprise_soa_pla

tform/5/html/esb_services_guide/universal_description_discovery_and_integration_uddi_registry.

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

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

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

[21] Twitter Diffy GitHub. https://github.com/twitter-archive/diffy, .

[22] Twitter Infrastructure. https://www.slideshare.net/InfoQ/decomposing-twitter-adventures-in-serviceoriented-architecture, .

[23] gRPC Website. https://grpc.io/, .

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

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

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

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

[28] 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.

[29] 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. doi: 10.1109/ICSA.2018.00014.

[30] Anonymous. Loghub, September 2021. URL https://doi.org/10.5281/zenodo.3227177.

[31] Florian Auer, Valentina Lenarduzzi, Michael Felderer, and Davide Taibi. From monolithic systems to microservices: An assessment framework. *Information and Software Technology*, 137:106600, 2021. ISSN 0950-5849. doi: https://doi.org/10.1016/j.infsof.2021.106600. URL https://www.sciencedirect.com/science/article/pii/S0950584921000793.

[32] Martin Aumller, Erik Bernhardsson, and Alexander Faithfull. Ann-benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. *Information Systems*, 87:101374, 2020. ISSN 0306-4379. doi: https://doi.org/10.1016/j.is.2019.02.006. URL https://www.sciencedirect.com/science/article/pii/S0306437918303685.

[33] Ricardo Ávila, Raphaël Khoury, Richard Khoury, and Fábio Petrillo. Use of security logs for data leak detection: A systematic literature review. *Security and Communication Networks*, 2021:6615899, Mar 2021. ISSN 1939-0114. doi: 10.1155/2021/6615899. URL https://doi.org/10.1155/2021/6615899.

[34] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. Microservices architecture enables devops: Migration to a cloud-native architecture. *IEEE Software*, 33(3):42–52, 2016. doi: 10.1109/MS.2016.64.

[35] 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 255259. IEEE Press, 2019. doi: 10.1109/MSR.2019.00051. URL https://doi.org/10.1109/MSR.2019.00051.

[36] 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. ISBN 978-3-319-67262-5.

[37] Shraddha Barke, Christian Poelitz, Carina Negreanu, Benjamin Zorn, José Cambronero, Andrew Gordon, Vu Le, Elnaz Nouri, Nadia Polikarpova, Advait Sarkar, Brian Slininger, Neil Toronto, and Jack Williams. Solving data-centric tasks using large language models. In Kevin Duh, Helena Gomez, and Steven Bethard, editors, *Findings of the Association for Computational Linguistics: NAACL 2024*, pages 626–638, Mexico City, Mexico, June 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.findings-naacl.41. URL https://aclanthology.org/2024.findings-naacl.41.

[38] 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. URL http://arxiv.org/abs/1906.05013.

[39] 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. ISSN 1544-3566. doi: 10.1145/3418899. URL https://doi.org/10.1145/3418899.

[40] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda

Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020. URL `https://arxiv.org/abs/2005.14165`.

[41] 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. doi: 10.1109/MS.2018.2141026.

[42] Phillip Carter. *Observability, Meet Natural Language Querying with Query Assistant*, May 2023. `https://www.honeycomb.io/blog/introducing-query-assistant`.

[43] Phillip Carter. *All the Hard Stuff Nobody Talks About when Building Products with LLMs*, May 2023. `https://www.honeycomb.io/blog/hard-stuff-nobody-talks-about-llm`.

[44] Phillip Carter. *Improving LLMs in Production With Observability*, May 2023. `https://www.honeycomb.io/blog/improving-llms-production-observability`.

[45] Luiz Carvalho, Alessandro Garcia, Wesley K. G. Assuno, 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. doi: 10.1109/CESSER-IP.2019.00012.

[46] Lianping Chen. Microservices: Architecting for continuous delivery and devops. 03 2018. doi: 10.1109/ICSA.2018.00013.

[47] Datadog. *Datadog Index*, May 2023. https://docs.datadoghq.com/logs/log_configuration/indexes/.

[48] Thomas Davidson and Jonathan Mace. See it to believe it? the role of visualisation in systems research. In *Proceedings of the 13th Symposium on Cloud Computing*, SoCC '22, page 419428, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450394147. doi: 10.1145/3542929.3563488. URL https://doi.org/10.1145/3542929.3563488.

[49] Thomas Davidson, Emily Wall, and Jonathan Mace. A qualitative interview study of distributed tracing visualisation: A characterisation of challenges and opportunities. *IEEE Transactions on Visualization and Computer Graphics*, pages 1–12, 2023. doi: 10.1109/TVCG.2023.3241596.

[50] 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. doi: 10.1109/ICSA.2018.00012.

[51] Pradeep Dogga, Karthik Narasimhan, Anirudh Sivaraman, Shiv Kumar Saini, George Varghese, and Ravi Netravali. Revelio: Ml-generated debugging queries for distributed systems. *CoRR*, abs/2106.14347, 2021. URL https://arxiv.org/abs/2106.14347.

[52] Dynatrace. *Observability and Security Convergence: Enabling Faster, More Secure Innovation in the Cloud*, December 2022. https://www.dynatrace.com/info/reports/cio-observability-security/.

[53] Simon Eismann, Cor-Paul Bezemer, Weiyi Shang, Dušan Okanović, and André van Hoorn. Microservices: A performance tester's dream or nightmare? In *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, ICPE '20, page 138149, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450369916. doi: 10.1145/3358960.3379124. URL https://doi.org/10.1145/3358960.3379124.

[54] 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):4465, August 2018. ISSN 1542-7730. doi: 10.1145/3277539.3277541. URL https://doi.org/10.1145/3277539.3277541.

[55] 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 318, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362405. doi: 10.1145/3297858.3304013. URL https://doi.org/10.1145/3297858.3304013.

[56] 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 1933, New York, NY,

USA, 2019. Association for Computing Machinery. ISBN 9781450362405. doi: 10.1145/3297858.3304004. URL https://doi.org/10.1145/3297858.3304004.

[57] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayantara Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Yuan He, and Christina Delimitrou. Unveiling the Hardware and Software Implications of Microservices in Cloud and Edge Systems. In *IEEE Micro Special Issue on Top Picks from the Computer Architecture Conferences*, May/June 2020.

[58] 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 135151, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383172. doi: 10.1145/3445814.3446700. URL https://doi.org/10.1145/3445814.3446700.

[59] Yunfan Gao, Yun Xiong, Meng Wang, and Haofen Wang. Modular rag: Transforming rag systems into lego-like reconfigurable frameworks, 2024. URL https://arxiv.org/abs/2407.21059.

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

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

[62] Grafana Labs. Logql documentation, 2024. URL https://grafana.com/docs/loki/latest/query/. Accessed: October 16, 2024.

[63] Sara Hassan, Rami Bahsoon, and Rick Kazman. Microservice transition and its granularity problem: A systematic mapping study. 50(9):1651–1681, June 2020. doi: 10.1002/spe.2869. URL https://doi.org/10.1002/spe.2869.

[64] Pinjia He, Jieming Zhu, Zibin Zheng, and Michael R. Lyu. Drain: An online log parsing approach with fixed depth tree. In *2017 IEEE International Conference on Web Services (ICWS)*, pages 33–40, 2017. doi: 10.1109/ICWS.2017.13.

[65] Shilin He, Xu Zhang, Pinjia He, Yong Xu, Liqun Li, Yu Kang, Minghua Ma, Yining Wei, Yingnong Dang, Saravanakumar Rajmohan, et al. An empirical study of log analysis at microsoft. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1465–1476, 2022.

[66] Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding. In *International Conference on Learning Representations*.

[67] Zijin Hong, Zheng Yuan, Qinggang Zhang, Hao Chen, Junnan Dong, Feiran Huang, and Xiao Huang. Next-generation database interfaces: A survey of llm-based text-to-sql, 2024. URL https://arxiv.org/abs/2406.08426.

[68] Wenpin Hou and Zhicheng Ji. Comparing large language models and human programmers for generating programming code, 2024. URL https://arxiv.org/abs/2403.00894.

[69] Edward J Hu, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, et al. Lora: Low-rank adaptation of large language models. In *International Conference on Learning Representations*.

[70] 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 7691, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450386388. doi: 10.1145/3472883.3486994. URL https://doi.org/10.1145/3472883.3486994.

[71] Yintong Huo, Yuxin Su, Cheryl Lee, and Michael R. Lyu. Semparser: A semantic parser for log analytics. In *Proceedings of the 45th International Conference on Software Engineering*, ICSE '23, page 881893. IEEE Press, 2023. ISBN 9781665457019. doi: 10.1109/ICSE48619.2023.00082. URL https://doi.org/10.1109/ICSE48619.2023.00082.

[72] 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. doi: 10.1109/IPDPSW.2017.40.

[73] 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 152166, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383172. doi: 10.1145/3445814.3446701. URL https://doi.org/10.1145/3445814.3446701.

[74] 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. ISBN 9781450355841. doi: 10.1145/3190508.3190546. URL https://doi.org/10.1145/3190508.3190546.

[75] 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 *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 3450, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450350853. doi: 10.1145/3132747.3132749. URL https://doi.org/10.1145/3132747.3132749.

[76] Suman Karumuri. Taming spiky log volumes: Maintaining Real-Time log accessibility with kaldb. Singapore, June 2023. USENIX Association.

[77] Suman Karumuri, Franco Solleza, Stan Zdonik, and Nesime Tatbul. Towards observability data management at scale. *SIGMOD Rec.*, 49(4):1823, mar 2021. ISSN 0163-5808. doi: 10.1145/3456859.3456863. URL https://doi.org/10.1145/3456859.3456863.

[78] Joon-Seok Kim, Hamdi Kavak, Chris Ovi Rouly, Hyunjee Jin, Andrew Crooks, Dieter Pfoser, Carola Wenk, and Andreas Züfle. Location-based social simulation for prescriptive analytics of disease spread. *SIGSPATIAL Special*, 12(1):5361, July 2020. doi: 10.1145/3404820.3404828. URL https://doi.org/10.1145/3404820.3404828.

[79] Myunghwan Kim, Roshan Sumbaly, and Sam Shah. Root cause detection in a service-oriented architecture. *ACM SIGMETRICS Performance Evaluation Review*, 41(1):93–104, 2013.

[80] Holger Knoche and Wilhelm Hasselbring. Drivers and barriers for microservice adoption - a survey among professionals in germany. 14:1–35, 01 2019. doi: 10.18417/emisa.14.1.

[81] 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. doi: 10.1109/MS.2012.3.

[82] Grafana Labs. *Loki*. Grafana Labs, 2024. URL `https://github.com/grafana/loki`. Accessed: October 21, 2024.

[83] 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. URL `https://arxiv.org/abs/2103.00170`.

[84] 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 3651, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383172. doi: 10.1145/3445814.3446696. URL `https://doi.org/10.1145/3445814.3446696`.

[85] Modal Lbas. *Modal Labs*, 2024. `https://modal.com/`.

[86] Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, et al. Can llm already serve as a database

interface? a big bench for large-scale database grounded text-to-sqls. *Advances in Neural Information Processing Systems*, 36, 2024.

[87] 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. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2019.07.008. URL https://www.sciencedirect.com/science/article/pii/S0164121219301475.

[88] JinJin Lin, Pengfei Chen, and Zibin Zheng. Microscope: Pinpoint performance issues with causal graphs in micro-service environments. In *Service-Oriented Computing: 16th International Conference, ICSOC 2018, Hangzhou, China, November 12-15, 2018, Proceedings 16*, pages 3–20. Springer, 2018.

[89] Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics*, 12:157–173, 2024. doi: 10.1162/tacl_a_00638. URL https://aclanthology.org/2024.tacl-1.9.

[90] Logz.io. Leveraging log management for business intelligence, 2024. URL https://logz.io/blog/log-management-business-intelligence/. Accessed: October 15, 2024.

[91] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=Bkg6RiCqY7.

[92] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. *Characterizing Microservice Depen-*

dency and Performance: Alibaba Trace Analysis, page 412426. Association for Computing Machinery, New York, NY, USA, 2021. ISBN 9781450386388. URL https://doi.org/10.1145/3472883.3487003.

[93] 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. doi: 10.1109/APSEC48747.2019.00031.

[94] Zeyang Ma, An Ran Chen, Dong Jae Kim, Tse-Hsun Chen, and Shaowei Wang. Llmparser: An exploratory study on using large language models for log parsing. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400702174. doi: 10.1145/3597503.3639150. URL https://doi.org/10.1145/3597503.3639150.

[95] Jonathan Mace, Peter Bodik, Rodrigo Fonseca, and Madanlal Musuvathi. Retro: Targeted resource management in multi-tenant distributed systems. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 589–603, Oakland, CA, May 2015. USENIX Association. ISBN 978-1-931971-218. URL https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/mace.

[96] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. Pivot tracing: Dynamic causal monitoring for distributed systems. *Commun. ACM*, 63(3):94102, feb 2020. ISSN 0001-0782. doi: 10.1145/3378933. URL https://doi.org/10.1145/3378933.

[97] Chandra Maddila, Negar Ghorbani, Kosay Jabre, Vijayaraghavan Murali, Edwin Kim, Parth Thakkar, Nikolay Pavlovich Laptev, Olivia Harman, Diana Hsu, Rui

Abreu, and Peter C. Rigby. Ai-assisted sql authoring at industry scale, 2024. URL https://arxiv.org/abs/2407.13280.

[98] Yev Meyer, Marjan Emadi, Dhruv Nathawani, Lipika Ramaswamy, Kendrick Boyd, Maarten Van Segbroeck, Matthew Grossman, Piotr Mlocek, and Drew Newberry. Synthetic-Text-To-SQL: A synthetic dataset for training language models to generate sql queries from natural language prompts, April 2024. URL https://huggingface.co/datasets/gretelai/synthetic-text-to-sql.

[99] Amirhossein Mirhosseini, Sameh Elnikety, and Thomas F. Wenisch. *Parslo: A Gradient Descent-Based Approach for Near-Optimal Partial SLO Allotment in Microservices*, page 442457. Association for Computing Machinery, New York, NY, USA, 2021. ISBN 9781450386388. URL https://doi.org/10.1145/3472883.3486985.

[100] Ghulam Murtaza, Amir R Ilkhechi, and Saim Salman. Impact of gdpr on service meshes. URL http://cs.brown.edu/courses/csci2390/2019/assign/project/report/gdpr-service-meshes.pdf.

[101] Irakli Nadareishvili, Ronnie Mitra, Matt McLarty, and Mike Amundsen. *Microservice architecture: aligning principles, practices, and culture.* " O'Reilly Media, Inc.", 2016.

[102] 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 521530, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781605580791. doi: 10.1145/1368088.1368160. URL https://doi.org/10.1145/1368088.1368160.

[103] Arpit Narechania, Arjun Srinivasan, and John Stasko. Nl4dv: A toolkit for

generating analytic specifications for data visualization from natural language queries. *IEEE Transactions on Visualization and Computer Graphics*, 27(2): 369379, February 2021. ISSN 2160-9306. doi: 10.1109/tvcg.2020.3030378. URL http://dx.doi.org/10.1109/TVCG.2020.3030378.

[104] CNCF TAG Observability. *CNCF TAG Observability - DSLs YAML*, September 2024. https://github.com/cncf/tag-observability/blob/main/working-groups/query-standardization/dsls/dsls.yaml.

[105] OpenAI. *GPT-4 API*, May 2023. https://platform.openai.com/docs/models/gpt-4.

[106] Anelis Pereira-Vale, Eduardo B. Fernandez, Ral Monge, Hernn Astudillo, and Gastn Mrquez. Security in microservice-based systems: A multivocal literature review. *Computers & Security*, 103:102200, 2021. ISSN 0167-4048. doi: https://doi.org/10.1016/j.cose.2021.102200. URL https://www.sciencedirect.com/science/article/pii/S0167404821000249.

[107] Xinyu Pi, Bing Wang, Yan Gao, Jiaqi Guo, Zhoujun Li, and Jian-Guang Lou. Towards robustness of text-to-SQL models against natural and realistic adversarial table perturbation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2007–2022, Dublin, Ireland, May 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.acl-long.142. URL https://aclanthology.org/2022.acl-long.142.

[108] Rodolfo Picoreti, Alexandre Pereira do Carmo, Felippe Mendona de Queiroz, Anilton Salles Garcia, Raquel Frizera Vassallo, and Dimitra Simeonidou. Multi-level observability in cloud orchestration. In *2018 IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing, 16th Intl Conf on Pervasive Intelligence*

and Computing, 4th Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress(DASC/PiCom/DataCom/CyberSciTech), pages 776–784, 2018. doi: 10.1109/DASC/PiCom/DataCom/CyberSciTec.2018.00134.

[109] 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. ISBN 978-1-939133-19-9. URL https://www.usenix.org/conference/osdi20/presentation/qiu.

[110] Nitarshan Rajkumar, Raymond Li, and Dzmitry Bahdanau. Evaluating the text-to-sql capabilities of large language models, 2022. URL https://arxiv.org/abs/2204.00498.

[111] 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.

[112] Aaron Sanders. *Grafana Community Dashboards Dataset*, June 2023. https://huggingface.co/datasets/sandersaarond/Grafana-Community-Dashboards.

[113] Shouvon Sarker, Xishuang Dong, Xiangfang Li, and Lijun Qian. Enhancing llm fine-tuning for text-to-sqls by sql quality measurement, 2024. URL https://arxiv.org/abs/2410.01869.

[114] Torsten Scholak, Nathan Schucher, and Dzmitry Bahdanau. PICARD: Parsing incrementally for constrained auto-regressive decoding from language models. In

Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih, editors, *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 9895–9901, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.emnlp-main.779. URL https://aclanthology.org/2021.emnlp-main.779.

[115] Tim Schopf, Dennis N. Schneider, and Florian Matthes. Efficient domain adaptation of sentence embeddings using adapters. In Ruslan Mitkov and Galia Angelova, editors, *Proceedings of the 14th International Conference on Recent Advances in Natural Language Processing*, pages 1046–1053, Varna, Bulgaria, September 2023. INCOMA Ltd., Shoumen, Bulgaria. URL https://aclanthology.org/2023.ranlp-1.112/.

[116] 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.

[117] Vishwanath Seshagiri, Siddharth Balyan, Vaastav Anand, Kaustubh Dhole, Ishan Sharma, Avani Wildani, Jos Cambronero, and Andreas Zfle. Chatting with logs: An exploratory study on finetuning llms for logql, 2024. URL https://arxiv.org/abs/2412.03612.

[118] 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. URL http://www.vldb.org/pvldb/vol13/p1064-shastri.pdf.

[119] Yuri Shkuro, Benjamin Renard, and Atul Singh. Positional paper: Schema-first application telemetry. *SIGOPS Oper. Syst. Rev.*, 56(1):817, jun 2022. ISSN

0163-5980. doi: 10.1145/3544497.3544500. URL https://doi.org/10.1145/3544497.3544500.

[120] 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. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2018.09.082. URL https://www.sciencedirect.com/science/article/pii/S0164121218302139.

[121] Splunk. *Indexes, indexers, and indexer clusters*, May 2023. https://docs.splunk.com/Documentation/Splunk/9.0.4/Indexer/Aboutindexesandindexers.

[122] 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. doi: 10.1109/IISWC.2018.8573515.

[123] 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 733750, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450371025. doi: 10.1145/3373376.3378450. URL https://doi.org/10.1145/3373376.3378450.

[124] 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. ISBN 978-1-939133-08-3. URL https://www.usenix.org/conference/osdi18/presentation/sriraman.

[125] Akshitha Sriraman, Abhishek Dhanotia, and Thomas F. Wenisch. Softsku:

Optimizing server architectures for microservice diversity @scale. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, page 513526, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450366694. doi: 10.1145/3307650.3322227. URL https://doi.org/10.1145/3307650.3322227.

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

[127] 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. doi: 10.1109/MCC.2017.4250931.

[128] Davide Taibi, Valentina Lenarduzzi, and Claus Pahl. *Microservices Anti-patterns: A Taxonomy*, pages 111–128. Springer International Publishing, Cham, 2020. ISBN 978-3-030-31646-4. doi: 10.1007/978-3-030-31646-4_5. URL https://doi.org/10.1007/978-3-030-31646-4_5.

[129] Gemma Team, Morgane Riviere, Shreya Pathak, Pier Giuseppe Sessa, Cassidy Hardin, Surya Bhupatiraju, Léonard Hussenot, Thomas Mesnard, Bobak Shahriari, Alexandre Ramé, et al. Gemma 2: Improving open language models at a practical size. *arXiv preprint arXiv:2408.00118*, 2024.

[130] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothe Lacroix, Baptiste Rozire, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models, 2023.

[131] Ben Treynor, Mike Dahlin, Vivek Rau, and Betsy Beyer. The calculus of service availability. *Commun. ACM*, 60(9):4247, aug 2017. ISSN 0001-0782. doi: 10.1145/3080202. URL https://doi.org/10.1145/3080202.

[132] 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 13591363, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450329224. doi: 10.1145/2675133.2675148. URL https://doi.org/10.1145/2675133.2675148.

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

[134] 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. doi: 10.1109/MASCOTS50786.2020.9285946.

[135] Mario Villamizar, Oscar Garcs, 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. doi: 10.1109/ColumbianCC.2015.7333476.

[136] 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. doi: 10.1007/s11761-017-0208-y. URL https://doi.org/10.1007/s11761-017-0208-y.

[137] VoidQuark. *Parsing SSH Logs with Grafana Loki*, 2024. https://voidquark.com/blog/parsing-ssh-logs-with-grafana-loki/.

[138] 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.

[139] 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. ISBN 978-3-319-62407-5.

[140] 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. doi: 10.1109/ICDCS.2017.32.

[141] Yingying Wang, Harshavardhan Kadiyala, and Julia Rubin. Promises and challenges of microservices: an exploratory study. *Empirical Software Engineering*, 26(4):63, May 2021. ISSN 1573-7616. doi: 10.1007/s10664-020-09910-y. URL https://doi.org/10.1007/s10664-020-09910-y.

[142] Muhammad Waseem, Peng Liang, and Mojtaba Shahin. A systematic mapping study on microservices architecture in devops. *Journal of Systems and Software*, 170:110798, 2020. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2020.110

798. URL https://www.sciencedirect.com/science/article/pii/S0164121220302053.

[143] 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 2130, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450388191. doi: 10.1145/3457913.3457939. URL https://doi.org/10.1145/3457913.3457939.

[144] Anna Wiedemann, Nicole Forsgren, Manuel Wiesche, Heiko Gewald, and Helmut Krcmar. The devops phenomenon. *Communications of the ACM*, 62:8, 2019. URL https://cacm.acm.org/magazines/2019/8/238341-research-for-practice-the-devops-phenomenon/fulltext.

[145] Wikipedia. Cubit — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Cubit&oldid=1274375061, 2025. [Online; accessed 25-March-2025].

[146] Yang Wu, Yao Wan, Hongyu Zhang, Yulei Sui, Wucai Wei, Wei Zhao, Guandong Xu, and Hai Jin. Automated data visualization from natural language via large language models: An exploratory study. *Proc. ACM Manag. Data*, 2(3), May 2024. doi: 10.1145/3654992. URL https://doi.org/10.1145/3654992.

[147] Zhenyu Xu and Victor S. Sheng. Detecting ai-generated code assignments using perplexity of large language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 23155–23162, 2024. doi: 10.1609/aaai.v38i21.30361.

[148] Jiaxi Yang, Binyuan Hui, Min Yang, Jian Yang, Junyang Lin, and Chang Zhou. Synthesizing text-to-SQL data from weak and strong LLMs. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar, editors, *Proceedings of the 62nd*

Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 7864–7875, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.425. URL https://aclanthology.org/2024.acl-long.425.

[149] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, Brussels, Belgium, 2018. Association for Computational Linguistics.

[150] 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. doi: 10.1109/ICSA.2019.00014. URL https://doi.ieeecomputersociety.org/10.1109/ICSA.2019.00014.

[151] 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 167181, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383172. doi: 10.1145/3445814.3446693. URL https://doi.org/10.1145/3445814.3446693.

[152] Zheng Zhang, Hossein Amiri, Zhenke Liu, Liang Zhao, and Andreas Zuefle. Large language models for spatial trajectory patterns mining. In *Proceedings of the 1st ACM SIGSPATIAL International Workshop on Geospatial Anomaly Detection*, GeoAnomalies '24, page 5255, New York, NY, USA, 2024. Association

for Computing Machinery. ISBN 9798400711442. doi: 10.1145/3681765.3698467. URL https://doi.org/10.1145/3681765.3698467.

[153] 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 149161, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450360111. doi: 10.1145/3267809.3267823. URL https://doi.org/10.1145/3267809.3267823.

[154] Shuyan Zhou, Uri Alon, Sumit Agarwal, and Graham Neubig. CodeBERTScore: Evaluating code generation with pretrained models of code. In Houda Bouamor, Juan Pino, and Kalika Bali, editors, *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 13921–13937, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.emnlp-main.859. URL https://aclanthology.org/2023.emnlp-main.859.

[155] 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.

[156] 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. doi: 10.1145/3183440.3194991. URL https://doi.org/10.1145/3183440.3194991.

[157] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Dewei Liu, Qilin Xiang, and Chuan He. Latent error prediction and fault localization for microservice applications by learning from system trace logs. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 683–694, 2019.

[158] Jieming Zhu, Shilin He, Pinjia He, Jinyang Liu, and Michael R. Lyu. Loghub: A large collection of system log datasets for ai-driven log analytics. In *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*, pages 355–366, 2023. doi: 10.1109/ISSRE59848.2023.00071.

[159] Olaf Zimmermann. Microservices tenets. *Computer Science - Research and Development*, 32(3):301–310, Jul 2017. ISSN 1865-2042. doi: 10.1007/s00450-016-0337-0. URL `https://doi.org/10.1007/s00450-016-0337-0`.