Distribution Agreement

In presenting this thesis as a partial fulfillment of the requirements for a 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 in whole or in part in all forms of media, now or hereafter now, 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. I retain all ownership rights to the copyright of the thesis. I also retain the right to use in future works (such as articles or books) all or part of this thesis.

Nikhil Bapat April 10, 2023

Leveraging Distributed Tracing for Root Cause Localization in Microservice-Architectured Distributed Systems

by

Nikhil Bapat

Ymir Vigfusson
Adviser

Computer Science

Ymir Vigfusson

Adviser

Nosayba El-Sayed

Committee Member

Andreas Zufle

Committee Member

2023

Leveraging Distributed Tracing for Root Cause Localization in Microservice-Architectured Distributed Systems

By

Nikhil Bapat

Ymir Vigfusson

Adviser

An abstract of
a thesis submitted to the Faculty of Emory College of Arts and Sciences
of Emory University in partial fulfillment
of the requirements of the degree of
Bachelor of Science with Honors

Computer Science

2023

Abstract

Leveraging Distributed Tracing for Root Cause Localization in
Microservice-Architectured Distributed Systems
By Nikhil Bapat

Developers and administrators of distributed systems lack a tracing and diagnostic framework to quickly identify and address causes of performance regression. When attempting to diagnose an issue in a distributed system, developers and administrators can observe some poorly performing requests and collect distributed trace data from the services in the system, but they still struggle to parse the traces and localize the source of the problem. In this paper, we present a methodology that, given a set of traces, identifies and ranks system components that are most likely to be the cause of performance regression. This approach saves system administrators and developers time identifying root causes of poor performance and can more precisely pinpoint issues that may otherwise fail to be adequately diagnosed.

Leveraging Distributed Tracing for Root Cause Localization in
Microservice-Architectured Distributed Systems

By

Nikhil Bapat

Ymir Vigfusson

Adviser

A thesis submitted to the Faculty of Emory College of Arts and Sciences
of Emory University in partial fulfillment
of the requirements of the degree of
Bachelor of Science with Honors

Computer Science

2023

Acknowledgments

First and foremost, I want to thank Dr.Ymir Vigfusson for introducing me to this project. Dr.Vigfusson is the reason I was first inspired to pursue computer science research. Although the process was challenging and the project hit many roadblocks, Dr.Vigfusson stayed supportive and optimistic. Without his belief in me, I would never have been able to complete my honors thesis. I'd like to also thank my committee members, Dr. Nosayba El-Sayed and Andreas Zufle. Both of them provided great guidance and feedback on my project, as well as being incredible, kind members of the Emory CS community. Dr. El-Sayed was my professor for CS 171 my freshman year and is one of the reasons I decided to pursue a Computer Science major, and it's been a wonderful full-circle experience to have her on my honors committee.

I also want to thank Vishwanath Seshagiri, a Ph.D. student in the SimBioSys lab. Vish was my mentor on this project, and was instrumental in helping me gain the background knowledge to get the project started, as well as helping me each week to make sure I met my deadlines and kept the project going despite the challenges. This project could not be possible without him, and I'm very grateful for his mentorship and guidance. I'd like to thank Yazhou Zhang for helping me get set up with the necessary tooling for my experiments. I'd also like to give a big shout out to the entire SimBioSys lab for their support and kindness throughout this entire process.

Finally, I'd like to thank my family for their support and belief in me. They constantly helped me overcome mental blocks and gave me the confidence to keep going in this project, and I could not have done this without their love and support. I'd also like to thank both Aneeka Patel and Daniel Cooley for their support and solidarity throughout this stressful process, and for helping me stay motivated and inspiring me to never give up.

Contents

1 Introduction **1**

2 Motivation **5**

3 Design **9**

3.1 Problem Overview 9

3.2 Localization Metric 11

3.3 Example: Applying Localization Method to Mock System 12

3.4 Statistical Fault Localization Notation 14

3.5 Other Root Causes 16

4 Implementation **18**

4.1 Basic Implementation 18

4.2 Sampling 19

4.3 Simulating Retroactive Sampling 20

5 Experiments **21**

5.1 Experimentation Process 21

5.2 Latency Injection Process 22

5.3 Evaluation Metrics 23

5.4 Sampling 24

6 Results 26

7 Related Work 31

7.1 Distributed Traces . . . 31
7.2 Retroactive Sampling . . . 33
7.3 DeathStarBench . . . 33
7.4 Fault Localization . . . 34
7.5 Benchmarks . . . 35
7.6 Distributed System Outage Detection . . . 37
7.7 Bayes Factor . . . 38

8 Conclusion and Future Work 39

Bibliography 41

List of Figures

3.1 A Small Mock Microservice Distributed System 9

6.1 Heimdall Prediction Accuracy Under Sampling 28

6.2 Ochiai Prediction Accuracy Under Sampling 29

7.1 Sample Visualization of a Jaeger Trace from DeathStarBench 32

7.2 Architecture of DeathStarBench Social Network 34

List of Tables

3.1 Suspicion Ratios of Components in Mock System 14

6.1 Prediction Evaluation Against Various Localization Systems 26

6.2 Prediction Evaluation Across Fault Localization Methods 27

Chapter 1

Introduction

Large distributed services are essential to the modern economy, servicing billions of users and generating massive amounts of revenue for the companies that operate them. These systems are implemented as intricate webs consisting of a number of different microservices, and single end-user request may traverse through thousands of different services before it is complete. One such complex microservice system is Uber, whose backend consists of roughly 4,000 microservices and a total of about 40,000 unique APIs that can be called as part of servicing a user's request. [28]

Given the massive economic reliance on these systems, they are also subject to stringent requirements, as they are expected to deliver quick, reliable service with minimal to no errors or timeouts. Maintaining high availability and low latency are essential, as any downtime or delay leads to measurable business impact. An internal report by CDN services company Akamai found that even a mere 100 ms delay in load time decreases conversion rates by 7%, and 53% of mobile website users will leave a page if the load time exceeds three seconds [3]. Meanwhile, a 100ms delay at Amazon leads to a 1% drop in sales [11]. With such a large amount of money at stake, even small percentage dropoffs translate to a massive loss of revenue, so companies must take every possible measure to keep latency issues to a minimum.

When latency issues do arise in a service, operators must react as quickly as possible and mitigate the problem so the duration of the problem and the number of users affected is minimized. However, within a tangled web of services, it can be very difficult to identify which component of the system is causing the issue, particularly when under a time crunch. One strategy for accelerating the root causing process, and the focus of the thesis, is to localize the performance problem: identify which components (e.g. specific hardware, services, recent code changes) should be looked at first to identify and fix the true root causes of the problem. This localization can save operators time, as they are given a prioritization of where the issue is more likely to reside and where they should devote their attention and resources.

Distributed tracing systems are widely used to record traces of end-to-end requests in distributed systems. [27] A single end-to-end trace, corresponding to a specific request, contains comprehensive information about all the machines visited by the request. Having end-to-end information is useful, as it connects the work done for a single request across different machines and services, providing a more complete view of a request compared to individual machine logs. Traditionally, distributed tracing is used to provide singular traces for analysis, or to provide averages for common-case behavior. In this case, we leverage the information collected in distributed traces to find commonalities between traces that exhibit similar performance issues.

Given the lack of tools to localize faults and diagnose root causes, engineers are often unable to quickly identify and remediate the root issues. Instead, the current strategies used often lead to some failures not being properly root-caused, so the engineer is not able to get insights from the bug or target that issue more precisely. An internal study at Microsoft [10] analyzed the response to production incidents at Microsoft teams, and found that the root cause diagnosis process was responsible for the largest average delays in issue response and mitigation time. These performance issues have tangible economic impact, and thus companies look for ways to reduce

their response time and get services performing properly as quickly as possible. In order to expedite the process of root cause identification and debugging, we propose a fault localization methodology that ranks various possible services or code chunks as being most likely to contain the root issue, giving engineers a prioritized list of where to look first while debugging. In order to get the necessary information to create this ranking, we leverage distributed tracing.

We approach the fault localization problem using a method called Trace-Based Root Cause Localization, which identifies parts of the system most likely to be causing performance issues. To do so, we define variables that correspond one-to-one to the instrumented components of the distributed system. When the symptoms of a performance problem are detected, the model identifies variables that are most likely to be involved in symptomatic traces relative to healthy ones. Once the model has evaluated these variables and ranked them as highly associated with poorly performing traces, the system operator is given an indicator of which components of the system are likely associated with the issue, helping to localize the root cause of the issue.

To demonstrate this methodology, we design and create Heimdall, a system that implements Trace-Based Root Cause Localization. Heimdall ingests distributed traces and outputs a ranking metric indicating which components of the system are likely root causes. We subjected Heimdall to controlled experimental evaluation using the DeathStar Microservices Benchmark framework [9]. In our experiments, we induced delay into specific microservices within DeathStarBench to simulate performance problems that occur in microservice systems, and observed whether Heimdall could accurately identify the true causes as problematic and rank them highly. We also test Heimdall with a tracing framework that uses retroactive sampling [27], so that Heimdall is able to collect more problematic traces that exhibit poor performance, and thus is able to more clearly identify issues and improve its ranking of potential

problem areas.

In summary, my thesis makes the following contributions:

- We present the mathematical foundations of the Trace-Based Root Cause Localization methodology for fault localization in distributed systems.
- We discuss the use case and impact of using Trace-Based Root Cause Localization in practice and the possible use cases and benefits to systems operators.
- We design and implement Heimdall, a system that applies Trace-Based Root Cause Localization. Heimdall ingests distributed traces, and outputs a ranking metric indicating which components of the system are likely root causes.
- We test Heimdall using real trace data collected from known distributed systems benchmarks, and demonstrate that Trace-Based Root Cause Localization is practical to use and efficient.
- We design and run controlled experiments to evaluate the accuracy of the fault localization predictions generated by our Trace-Based Root Cause Localization approach under various sampling methodologies.
- We run Heimdall under various both head-based sampling and retroactive sampling, demonstrating the feasibility of Trace-Based Root Cause Localization under space constraints.

Chapter 2

Motivation

In recent times, distributed services have become increasingly vital to the global economy. To maintain high performance and low latency in these services, operators must be able to respond quickly to any issues that arise by identifying the underlying cause and fixing it. However, identifying the root cause of a problem can be a significant challenge, leading to delays in mitigating production issues. This issue is particularly pronounced in large, complex industry systems, where identifying root causes can be a significant challenge at scale and can cause significant delays in response time. A recent work [10] from Microsoft analyzed the response to production incidents in Microsoft Teams, and the various strategies used to diagnose and mitigate the issues. The study found that identifying complex root causes for production issues created lengthy delays, and in fact caused the longest delays in the process of mitigating production issues. In the majority of cases, this delay was due to engineers taking a long time to read through the code and identify and fix the bugs. Due to the difficulty of identifying root causes and bugs in code in a short amount of time, the Microsoft paper found that it was also very common for engineers to not directly fix the root cause. The study found that while roughly 40% of incidents were caused by code or configuration bugs, only about half of those were mitigated with a change to the

code or configuration. The most common strategy used was simply to rollback the recent changes to an older, working version. While this removes the bug and may be a quicker immediate fix, heavily relying on rollbacks rather than fixing the issue directly has its own drawbacks. Firstly, broadly rolling back recent commits removes other new features or changes that did not contribute to the bug, meaning that good, working code is not included in production. Secondly, failing to address the root cause runs the risk of a similar issue arising in the future, as the engineer may reimplement the same buggy ideas. In fact, in 35% of Microsoft Teams incidents, there was not a completed postmortem report detailing the issue, root cause, and mitigation, which can lead the team to lack information about different kinds of incidents that could arise going forward.

To address this concern and ensure that more incidents are properly root-caused and mitigated, there is a need for tools that can reduce the amount of time taken to find the root cause of an issue. Issues in distributed systems can range in severity, anywhere from small performance bumps, to individual request failures, to a complete outage of the entire system. However, this thesis focuses specifically on identifying the root causes of performance regressions, i.e. cases where the system performs exceptionally slowly or poorly utilizes resources. While these may be less notable issues than complete system outages, performance metrics such as latency are still of critical importance, and companies are deeply invested in their improvement. Therefore, a tool that can help operators identify the root cause of performance regressions more quickly can be of great importance for companies using microservice.

Distributed traces have been growing in popularity for large distributed systems as a way of tracing requests end-to-end and getting the necessary information about the system, which can then be utilized for a variety of diagnostic and troubleshooting tasks. However, the adoption and utilization of distributed traces among industry systems in production has been hampered by a lack of knowledge of how to use

distributed traces, a steep learning curve to properly set up and utilize tracing frameworks, and a significant amount of tedious manual work needed to extract insights from the trace files [23] [6] [4]. A report from Dynatrace [7] suggests that the distributed traces must be complemented by tools built on top of the traces that allow users to more easily extract the insights from the traces in an easy to consume manner. Such tools can make using distributed tracing more accessible to industry system operators and engineers, and help them utilize the technology to its full potential and gain deeper insights about the traces and their system.

Currently, distributed tracing is not heavily utilized as a method of localizing and identifying issues, particularly with regards to software bugs [6]. Therefore, a method like Trace-Based Root Cause Localization that can ingest distributed traces to find root causes and commonalities among poorly performing traces has a wide variety of novel use cases in a large distributed system. Its first use case is for an on-call engineer or system administrator trying to respond to a reported spike in latency on a system in a production environment. When responding to an outage, speed is paramount, and the respondent must be able to quickly diagnose the problem before it can be fixed. In a large system, the possibilities for where the issue might lie are massive, so a tool that reduces the potential problem space into ranked likelihoods can help the operator search and identify where the issue actually lies.

Trace-Based Root Cause Localization can also be used to identify edge case performance issues that are not immediately obvious from an aggregate analysis. Even if the actual additional latency is not very high, Trace-Based Root Cause Localization could detect that all traces in the 90th percentile or higher for latency share similar characteristics, or pass through the same specific nodes. This might be a sign of a mild performance issue in that node, and remediating that issue early might prevent future spikes, as well as improve performance consistency and overall runtime in the system. This is especially important because in addition to performance optimiza-

tion, system operators also want to see strong performance stability with predictable service performance at an acceptable level being preferred over a wild ebb and flow in performance.

Trace-Based Root Cause Localization is also useful for benchmarking multiple versions of a service against each other, which can be useful both in development when experimenting and testing new versions as well as for diagnostic benchmarking and comparison in a production environment. Such a use case is akin to A/B testing, either before a new version is deployed in an experimental framework, or in production as a "continuous" A/B test. Given various versions of a service that are present in a system, and a load balancer or other logic that feeds requests to one of these versions, Heimdall can be used to identify whether one or more of the versions is more associated with poorly performing traces. Such an analysis might uncover bugs or performance faults in one specific version, that could then be patched.

A survey of industry participants [23] who use microservice-architectured systems found that adoption of distributed traces has been hampered because companies are not sure how to adequately leverage them, and could not put in the necessary time to set up and understand tracing systems. Another respondent noted that distributed tracing was only useful to them "months in or years [later when] something's gone really wrong somewhere and you're trying to figure out who broke it". Other respondents from various companies shared that they were dissatisfied with the amount of testing. The more tools that exist that allow users to interface with the insights from their traces without having to learn the details of the tracing frameworks directly, the more accessible distributed tracing will become. Trace-Based Root Cause Localization fills these gaps and shortcomings and provides an accessible way to extract insights from the trace data and apply it to improve system performance.

Chapter 3

Design

3.1 Problem Overview

As a small illustrative example, consider Figure 3.1 as a miniature distributed system, where each node is representative of some microservice that handles the request. Requests first pass through at Node 1 and node 2, before being being directed to either node 3a or 3b based on some load balancing or other choice mechanism. Then, the request is complete upon arriving at node 4.

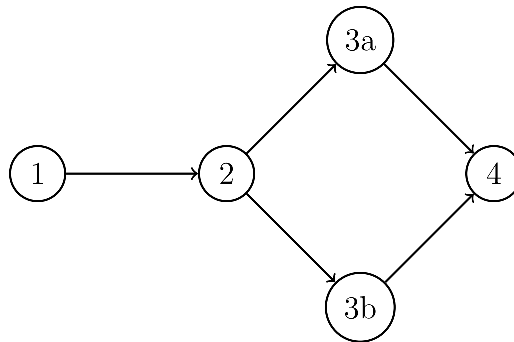

Figure 3.1: A Small Mock Microservice Distributed System

However, now suppose some performance issue occurs in node 3b (perhaps a new buggy version of the service that leads to unnecessary added delay was pushed to

only one machine, or there was an issue with the hardware). Therefore, all future requests that are serviced by node 3b will experience a spike in latency, while the requests directed through node 3a will perform as expected with an adequately low latency.

Therefore, all future requests that pass through node 3b should experience unusually high end-to-end latency compared to requests directed through node 3a. Identifying these poorly performing requests and flagging them as problematic can be done by a fairly quick analysis of the distributed traces (finding the total duration of each request, and then flagging some top percentile of them). However, the true challenge is identifying *why* these requests are slow, the true root cause of the latency.

To narrow down the true root cause, the operator of the system is in need of some method or heuristic that identifies the actual problem area (in this case node 3b) as a strong suspect for the root cause, so that the operator is pointed in the right direction to diagnose and mitigate the problem quickly. Given the available distributed traces for both slow and successful requests, we can attempt to use the traces as a set of clues that we can analyze to narrow down and localize the true source of the problem.

We then must decide how to methodically utilize the information in the traces to find the true root cause. One approach is to simply find the commonalities between all the slow requests, naively ranking the system components with respect to the number of poorly performing requests that they services. However, this runs into an obvious problem in the above graph, where all the poorly performing requests are not only serviced by the correct root cause node 3b, but also by the healthy nodes 1, 2, and 4. Ranking the nodes in order of the number of poorly performing requests fails to single out node 3b in any meaningful way. Instead, what the operator needs is a metric that can pick out the components or variables that are most associated with poorly performing traces, and subsequently less likely to be associated with healthy traces.

3.2 Localization Metric

In the search for an appropriate metric to accurately identify the faulty component, we need to highly weight association with poorly performing traces, while giving lower rankings to components and variables associated with healthy traces. For this, we set up a metric we call a suspicion ratio or odds-ratio. Somewhat inspired by the Bayes Factor [19], our ratio compares the support for two competing hypotheses: our "null hypothesis," that the component or variable in question is not the true root cause of the observed regression, and the "alternative hypothesis," that the particular component is the true root cause of the regression. What we seek to evaluate is the ratio between how much the data supports the notion that the component is associated with performance regression and how much it supports the notion that it is more associated with healthy traces.

Before fully defining our suspicion ratio σ_x, we must first formally define the components that make up our definition. For any trace T, we say that $x \in T$ if the x was included in the trace. In the case of x representing system nodes or microservices, as it does for the majority of our analysis, this means that the particular service or services were executed as part of servicing the request. If x refers to some other characteristic (see Section 3.5), then $x \in T$ might be interpreted as the variable being a part of the original request.

We also use F_T to represent whether a trace T was flagged as being poorly performant. For the case of our analysis, we flag poorly performing traces as being in some top percentile of latency, however the classification of flagged traces can be defined in any way according to the needs of the operator. If $F_T = 1$, we say that trace T was flagged for demonstrating poor performance. If $F_T = 0$, then trace T was not flagged for poor performance, which we also often refer to as healthy traces.

Once we have defined all the subparts of our suspicion ratio, we can use them to construct our overall localization metric. Formally, we express σ_x, our suspicion ratio

for some system component or variable x, with the following formula:

$$\sigma_x = \frac{P(x \in T | F_T = 1)}{P(x \in T | F_T = 0)}$$

In the numerator, we have the likelihood that that a flagged trace includes the variable or component in question, and the denominator is the likelihood that a healthy trace contains the variable, based on the trace data collected. This metric addresses and problems and meets the needs in Section 3.1, as it increases the score when a variable is closely linked with flagged traces, but conversely decreases the score for variables associated with healthy traces, avoiding the case where a variable associated with many or even all traces is assigned a high score despite being unproblematic.

The formula does encounter an edge case if $P(x \in T|F_T = 0) = 0$, i.e. there are no traces where the variable x is present and the trace is healthy. In this case, we can just assign the score as $\sigma_x = \infty$, as having no relation to any healthy traces is a good sign that the component is problematic, and we want it to appear at the top of our rankings.

3.3 Example: Applying Localization Method to Mock System

Now that we have a definition of our localization metric, we can begin to test it, first using the mock system drawn in Figure 3.1. Imagine that there are 100 requests that pass through the system, with random load balancing after node 2 that splits requests evenly between node 3a and 3b. Let the first 90 requests pass through without any issue, with 45 requests being services by 3a and 3b each. However, between the completion of request 90 and the beginning of request 91, some unspecified issue occurs in node 3b that causes a spike in latency for any request that passes through

that node. Thus, in the next 10 requests, of the 5 that pass through node 3b, all experience high latency.

Now let's assume that we define the threshold for flagging poorly performing traces as being in the top 10%, or the 90th percentile, of traces in terms of total end-to-end latency. In this case, the 10 flagged traces include the 5 problematic traces that passed through node 3b after the latency injection, as well as the 5 slowest traces among the 90 that were completed before the issue arose. For this mock experiment, we say that 3 of those requests were serviced by node 3a, and 2 by node 3b. (Note that while the 5 flagged traces don't actually suffer from the big issue at hand, our threshold still includes them, as in reality we are blinded to the number of truly symptomatic traces affected by an issue. Given the defined threshold, we simply take the traces that fall under that definition and find commonalities).

Overall, we find that of the 10 problematic traces, 3 were serviced by node 3a, and 7 by node 3b. We also notice that all 10 of these traces were serviced by nodes 1, 2, and 4. Similarly, all 90 of the healthy traces were serviced by nodes 1, 2, and 4, while node 3a serviced 47, and node 3b serviced 43.

For the sake of our example, we walk through the calculation of the suspiciousness ratio for each component. For nodes 1, 2, and 4, we calculate them together, as the set of traces executed by each component is identical. Regardless of whether the trace was triggered or not, the probability that the component serviced the trace is 100%, as these components serviced every trace. Thus, we find that $\sigma_1 = \sigma_2 = \sigma_4 = 1/1 = 1$

For node 3a, of the 10 flagged traces, node 3a serviced 3 of them. Thus, we find from our data that $P(3a \in T|F_T = 1) = 3/10$. Of the 90 healthy traces, node 3a serviced $50 - 3 = 47$ of them, so $P(3a \in T|F_T = 0) = 47/90$. When we compare the ratios, we find that $\sigma_{3a} = \frac{3/10}{47/90} \approx 0.574$.

For node 3b, of the 10 flagged traces, node 3b serviced 7 of them. Thus, we find from our data that $P(3b \in T|F_T = 1) = 7/10$. Of the 90 healthy traces, node 3b

Rank	Node	Ratio
1	3b	1.465
T2	1	1
T2	2	1
T2	4	1
5	3a	0.574

Table 3.1: Suspicion Ratios of Components in Mock System

serviced $50 - 7 = 43$ of them, so $P(3b \in T | F_T = 0) = 43/90$. When we compare the ratios, we find that $\sigma_{3b} = \frac{7/10}{43/90} \approx 1.465$.

Taking these ratios and ordering them by their score, we can generate the ranked list of the possible root cause locations in Table 3.1, giving a system operator or engineer a treasure map of where to look first for the issue. Following this list, they are directed to look at node 3b first, saving them time that they otherwise might have spent first looking at other nodes or possible issues. Also note that our example case contains a tie in the rankings. In this case, nodes 1, 2, and 4 are unable to be meaningfully differentiated, so there is no way to break the tie. We don't propose an explicit tie-breaking mechanism, since with a larger number of requests and systems with more complex branching, ties are very uncommon. If the scores are actually perfectly identical in practice, we recommend that the operator check both variables and assess them as equally suspicious, or perhaps use their own judgement with information not in the traces (perhaps some component or variable is known to cause issues often).

3.4 Statistical Fault Localization Notation

We have defined our odds ratio using probability notation, more similar to the Bayesian statistical methods used for hypothesis testing. However, we can also express this metric with a definition that uses fault localization notation from [21], which can make it easier to experiment and evaluate against other methods and metrics for

statistical software fault localization.

Under this notation, let s represent a variable or set of variables present in some subset of traces. Members of this set might include specific nodes or services a request passes through, but could also be extended to include other information about the request, such as which particular version or commit of the service is running, the system and hardware specifications of the machines servicing the request, or the locale from which the request originated.

Once we have identified our set s, we can divide all traces into one of four classifications — a^s_{ep}, a^s_{ef}, a^s_{np}, and a^s_{nf}. The subscript e represents that the set s exists in the trace, whereas the subscript n represents that s does not exist in the trace. The second subscript refers to whether the request performed properly or not. In this case, we use p to represent that a request demonstrated proper, healthy performance or f if it was flagged or triggered for poor performance. Therefore, a^s_{ep}, for example, represents the number of traces where the qualities of set s are present and the trace performed properly.

Statistical fault localization has been utilized across a variety of fields, and has already yielded a variety of different existing localization measures. [22] [2] [12][14] [26] While each of these metrics was developed for a slightly different localization use case, we can use them as a comparative benchmark to our suspicion ratio, and both assess how our new metric compares to existing standards and discern any shortcomings or needed improvements. Below we list four of the most popular of the existing metrics [24] using our above notation:

$$Ochiai : \frac{a^s_{ef}}{\sqrt{(a^s_{ef} + a^s_{nf})(a^s_{ef} + a^s_{ep})}} \qquad Tarantula : \frac{\frac{a^s_{ef}}{a^s_{ef}+a^s_{nf}}}{\frac{a^s_{ef}}{a^s_{ef}+a^s_{nf}} + \frac{a^s_{ep}}{a^s_{ep}+a^s_{np}}}$$

$$Zoltar : \frac{a^s_{ef}}{a^s_{ef} + a^s_{nf} + a^s_{ep} + \frac{10000 a^s_{nf} a^s_{ep}}{a^s_{ef}}} \qquad Wong-II : a^s_{ef} - a^s_{ep}$$

Now that we have proper definitions of the other metrics, we define our new metric using the same notation. This is for ease of experimentation and evaluation, as it allows us to more easily compare and analyze the results across the various metrics with the same experimental data in the same format. With the notation above, the suspicion ratio used in Heimdall is expressed as:

$$\sigma_x = \frac{\frac{a^s_{ef}}{a^s_{ef}+a^s_{nf}}}{\frac{a^s_{ep}}{a^s_{ep}+a^s_{np}}}$$

3.5 Other Root Causes

In our design and implementation of Heimdall, we focus on a simpler, though still common, case of performance issues, where the performance regression occurs due to an single, isolated issue, that is part of a singly particular service or machine. These single-source performance regressions could be caused by a variety of issues, spanning both hardware and software. For example, the same spike in latency might be due to a new, buggy version of a service that has an abnormally long runtime duration, network issues in a particular physical data center, or issues with caching or memory access on a specific machine, just to name a few. However, since the aim of Heimdall is simply to help localize the source of the fault, rather than directly diagnose the root cause on its own, the actual cause of the latency is not relevant to Heimdall. In all of these cases, Heimdall simply aims to identify a rough idea of the root cause or the area where the issue stems from, which in our implementation and testing case is a specific node running a version of a service.

However, the logic of Heimdall can be extended to cases beyond single-node performance issues. For example, performance problems in a system might arise when two particular services are active in the same timeframe working on the same request, but not when just one service or the other is executed. This might be because the

two services have shared resources, and when they both run at the same time, the system experiences slowdown. Another possibility is the presence of data format or compatibility bugs in particular versions of software, identified in [18] as a common issue in distributed systems. Regardless of the detailed reason for the issue, a framework that only identifies single nodes may not be effective in mitigating the issue, as a detailed troubleshooting of only that service might not yield any breakthroughs in improving performance. However, if Heimdall identifies the potential issue area s not as one service or the other, but as the set of the two services, the insight is clearer that the two services together, not one or the other, is more likely to cause the issue.

Other possible causes for performance regression could be related not directly to the actual services and machines that run to service the request, but characteristic about the trace itself. While these may not always correlate directly to simple software fixes, the ability to identify them can still be useful to companies managing large distributed systems. For example, we might let s in the analysis to represent requests that originate from a specific geographical region. If the ranking for a geographic region is high in the Heimdall rankings, this indicates that requests from that region are strongly associated with poor performance. This might be indicative of a lack of adequate service or coverage in that region, and a company might be motivated to construct more data centers nearer to that area, or improve load balancing, in response to this insight.

Chapter 4

Implementation

4.1 Basic Implementation

Our implementation of Heimdall begins with the ingestion of distributed traces collected by some distributed tracing system. The traces contain information linking requests to the individual services or service versions that serviced them. This information can be linked across services because the distributed tracing paradigm is intended to trace a request end to end, so the unique identifier for the trace is maintained throughout.

The input must also contain some sort of indicator for what traces are considered problematic or symptomatic. This is up to the user to designate their own thresholds and criteria for poorly performing traces, although the usual use case would be to designate problematic traces as being in the top k% of latency or some other performance metric. For the purpose of our implementation and testing, we set the bar for problematic traces to be the traces in the top 1% for end to end latency.

Once we have established relationships between traces and the services that they used, and a list of triggered traces, we can find the four component metrics identified in the above section, a^s_{ep}, a^s_{ef}, a^s_{np}, and a^s_{nf}. From these counts, we are then able to

compute the suspicion ratios as well as the various other fault localization metrics in Section 3.4 and sort the components in order of their scores to find the issue localization rankings, provided a guided map to the system operator of where to look for the issue.

4.2 Sampling

It is highly impractical to store traces for every request that passes through a system, due to the large storage overhead when the number of requests scales. As such, distributed tracing tools must use some sampling methodology, deciding which traces end up stored in the long term. Most traditional distributed tracing tools, including Jaeger, use a head-based sampling, meaning that a small percentage of requests are randomly selected to be stored. However, this sampling decision occurs at the beginning of the request, meaning that the sampling does not know whether or not the request may end up demonstrating performance problems. This poses a problem when attempting to diagnose poorly performing requests. If the poorly performing requests, which might make up only a small percentage of cases, are not sampled and stored, there is no way to use them as part of a fault localization analysis, as no traces exist as record of their performance. As such, use of traces for root cause localization has previously been thought to be unfeasible for high-scale industry systems with large numbers of requests, as they cannot possible store every request and head-based sampling does not produce satisfactory results.

To address this shortcoming of head-based sampling, Heimdall can benefit from the use of retroactive sampling, proposed in [27]. Under retroactive sampling, all traces are temporarily stored in a buffer at each node. If the request reaches completion and is then flagged as problematic due to high latency, then the trace gets stored for the long term. If the request is healthy, then the trace is not stored, and

can then be deleted or overwritten. The use of buffers and retroactive sampling is also combined with head-based sampling to keep some percentage of healthy traces as well. By using this combination, it is possible to get a full view of more problematic traces, as well as some picture of the healthy traces, all while maintaining a storage overhead that is relatively low.

4.3 Simulating Retroactive Sampling

For this project, the DeathStarBench microservice benchmark system used for experimentation was implemented using Jaeger, which collects traces under head-based sampling. However, even with a tracing framework that does not naturally support retroactive sampling, we are able to implement a subsampling method to simulate the effects of retroactive sampling and evaluate the effects of this sampling approach on our predictive rankings compared to head-based sampling.

To simulate the retroactive sampling method, we begin by running Jaeger with a 100% head-based sampling rate, which collects traces for every single request, both symptomatic and healthy. After the traces are collected and the poorly performing requests are flagged, we can randomly select a small percentage of the healthy traces to be kept, and a large percentage of the problematic traces to be kept. Thus, we simulate the dataset that we would get if we were to use a retroactive sampling tracing tool such as Hindsight, with a full view of all the poorly performing traces, and a small subset of the healthy traces. By running analysis on this reduced dataset, we are able to see the effects that retroactive sampling has on the strength of the Heimdall predictions, and can compare it to the accuracy of pure head-based sampling for a similarly sized sample.

Chapter 5

Experiments

5.1 Experimentation Process

In order to evaluate the predictions generated by Heimdall, we subject it to a series of tests in which a latency issue is injected into a benchmark service, and Heimdall, blinded to the true location of the performance issue, uses the generated distributed traces to evaluate and rank the possible problem nodes. Since the goal of Heimdall is to save the system administrator time in diagnosing and root-causing the issue, Heimdall is considered more successful the higher it ranks the true root cause. Thus, we evaluate the Heimdall rankings by whether it ranked the true cause sufficiently high, which we define using both a top-k metric, i.e. the percentage of cases in which the true root cause scored in the top k in the prediction rankings, and the mean reciprocal rank, or MRR, of the prediction. More information on the evaluation metrics can be found in Section 5.3.

For the experiments, we followed the following process:

- Send requests to the DeathStarBench social network system, an established microservice benchmark, with latency injected into one specific version of one service using the usleep command.

- Collect and store traces from DeathStarBench instrumented with the Jaeger distributed tracing framework.
- Feed the distributed traces into Heimdall, which will use Trace-Based Root Cause Localization to generate a ranking of services predicted to be the true cause. Note that Heimdall only takes the Jaeger traces as input, but is blinded to the actual service that was injected with extra latency.
- Compare the ranking generated by Heimdall with the actual service that was given added latency, and calculate the hitrate and MRR evaluation metrics, and compare them to alternative methods and related work.

5.2 Latency Injection Process

The latency injection process into a specific service is designed to be a controlled simulation of some latency-causing error that might occur within a service. The possible causes of such latency are numerous, although Heimdall cares only where the issue occurs, not what it actually is. Therefore, we find that it is sufficient to use any method to inject a set amount of latency into a machine of our choosing, so long as we can observe this latency in the trace files for that service as well as potentially any upstream services.

We simulate latency in our service using is injected using the *usleep* Linux command [20], which sets the program to sleep for a specified microsecond value. To configure this in a way that only affects one version of one service, we can build two separate docker images for the symptomatic service being experimented on, one with a call to *usleep* and one without. This is analogous to a software bug, that causes the program runtime to be anomalously longer than expected

To set this up for our experiments, we create two versions of every service, and make sure to host them on separate nodes in the docker swarm. Then we select the

service we wish to experiment on, and inject latency into one of the versions by adding a *usleep* command. For our experiments, we used *usleep(5000)*, which causes 5000 microseconds, or 5 milliseconds, of delay. We then built the two versions into two separate docker images under the same image name, which allowed the service to be replicated across two different nodes running slightly different versions of the service (one with delay, one without).

5.3 Evaluation Metrics

To evaluate the Heimdall rankings relative to the ground truth of the true cause, we turn to two kinds of evaluation metrics. The first style of evaluation uses top-k hitrate, also written HR@k, for some chosen integer k. Such a metric is formally defined [17] as:

$$HR@k = \frac{1}{|A|}\sum_{i=1}^{|A|}(r_i \in Rank_i^k)$$

where $|A|$ represents the total number of issues (i.e. the total number of experiments run in our case), r_i is the true root cause of the ith issue, and $Rank_i^k$ is the list of top-k ranked outputs in the ranked result list.

Put simply, this type of metric evaluates how often the true root cause was ranked among the top k possible root causes in the ordered ranking. Specifically, we chose to look at the top-1, top-3, and top-5 hitrates, which is in line with existing systems that we can benchmark against.

The second type of evaluation metric is the mean reciprocal rank, or MRR. Mean reciprocal rank is defined by [17] as the multiplicative inverse of the rank of the correct answer, or formally as:

$$MRR = \frac{1}{|A|}\sum_{i=1}^{|A|}\frac{1}{Index_i}$$

where $Index_i$ is the numerical placement of the true root cause among the list of ranked outputs, and all other component are defined the same as in the top-k case.

The range of the MRR metric ranges between $1/n$ and 1, where n is the number of possible root causes. The formal definition also contains the caveat that if the correct root cause does not exist in the rankings, then the MRR is infinity; however this cannot happen in Heimdall as every component is evaluated and included in the rankings.

5.4 Sampling

Both the sampling rate and the type of sampling used by the distributed traces inputted into Heimdall are important variables to consider when subjecting Heimdall to experimentation and evaluation. While Heimdall might be able to accurately rank problem areas when it has access to all the traces for all the requests, it is important for practical applications that Trace-Based Root Cause Localization is able to work with a smaller subset of traces, that is more manageable to store. With Jaeger, this is by default done using head-based sampling, though retroactive sampling can also be simulated as outlined in section Section 4.3.

To test the accuracy of Heimdall under different sampling rates and methodologies, we can subject it to a series of tests. Under head-base sampling, the sampling rate is a modifiable parameter in the Jaeger configuration files; we can also take a sample by a simple random sample of our dataset at the a specified sampling rate. Adjusting this sampling rate at various levels, we can evaluate how a change in sampling rate affects the Trace-Based Root Cause Localization prediction quality, and specifically the magnitude of the dropoff as the sampling rate, and therefore trace storage overhead, decreases.

When simulating retroactive tracing with Jaeger, the methodology has a few more

steps. Since the reason for sampling in the first place is to keep the space used for storing traces relatively low, we should compare the two sampling methodologies in a way that keeps the total traces stored equal. For the case where the head-based sampling is 100%, the approaches are equivalent, since all traces are captured. However, when the head based sampling rate is reduced, we adjust the random sampling in our retroactive sampling simulation to get an equal number of traces. Doing so depends on how we define symptomatic traces, and the sampling rate desired for symptomatic traces.

For the purpose of this example, say that we define symptomatic traces as the top 1% of requests in terms of latency, and we want to collect 100% of symptomatic traces. Additionally, we are trying to match the space footprint of the experiment with a head-based sampling rate of 10%, i.e. randomly selecting 10% of all traces. To simulate this beginning with our dataset of all traces sampled at 100%, we start by taking the 1% of traces marked as symptomatic, and then pick 1/11, or 9.09%, of the healthy traces. This collects a total of $1\% + 99/11\%$, or 10% of total traces, so the space taken by the two sampling methodologies is equal. Thus, under equal resource constraints, we can then evaluate the efficacy of the sampling methods in generating accurate fault prediction rankings and see if a change in sampling methodology reduces the dropoff in prediction accuracy as sampling rate decreases.

Chapter 6

Results

After collecting traces from the DeathStarBench social network using the first few steps of the process in Section 5.1, we were able to input these traces into Heimdall and generate predictions of the true root cause service. Heimdall is blinded to the root cause service, and must ascertain a ranking from the traces. Once the ranking is generated, we can compare it to the true root cause and compute the evaluation metrics outlined in Section 5.3.

We then compared the predictions generated by Heimdall against previous works in fault localization, comparing our Trace-Based Root Cause Localization approach against alternative strategies that did not utilize distributed tracing. In Table 6.1, we see that the odds ratio method scored top-1, top3, and top-5 hitrates of 0.50, 0.79, and 0.83 respectively, with an MRR of 0.65, and the Ochiai method scored 0.62, 0.83, 0.84, and 0.73 respectively, both outperforming the previous works across all four evaluation metrics.

Metric	Odds Ratio	Ochiai	MicroHECL	MonitorRank	Microscope
HR@1	0.50	0.62	0.48	0.32	0.35
HR@3	0.79	0.83	0.67	0.40	0.49
HR@5	0.83	0.84	0.72	0.43	0.59
MRR	0.65	0.73	0.58	0.37	0.44

Table 6.1: Prediction Evaluation Against Various Localization Systems

Metric	Odds Ratio	Ochiai	Tarantula	Zoltar	Wong-II
HR@1	0.50	0.62	0.50	0.29	0.21
HR@3	0.79	0.83	0.79	0.31	0.37
HR@5	0.83	0.84	0.83	0.34	0.50
MRR	0.65	0.73	0.65	0.35	0.35

Table 6.2: Prediction Evaluation Across Fault Localization Methods

We also attempt to compare the "suspiciousness ratio" against previously existing fault localization metrics from Section 3.4. In this table, all methods use the same set of distributed traces generated from the same set of requests, where the only difference is the way in which the a^s_{ep}, a^s_{ef}, a^s_{np}, and a^s_{nf} components are used to generate a metric. In Table 6.2, we observe that our ratio outperforms both the Zoltar and Wong-II metrics, performs roughly equivalently with the Tarantula method, and is outperformed by the Ochiai method.

To gain even further insight into these evaluation metrics, we can evaluate the statistical significance of the difference in proportions. Specifically focusing on the HR@1, we use a 2-proportion significance with $n = 197$ for both proportions. We find that Heimdall performs significantly better than both the Zoltar and Wong-II methods with $p = 6.3 * 10^{-6}$ and $p = 1.4 * 10^{-10}$ respectively, and the Ochiai method performs significantly better than Heimdall with $p = 7.8 * 10^{-3}$.

Given that the Ochiai metric performed significantly higher than any other metric in the evaluation, we include Ochiai in our analysis and experiments going forward, and it should be considered when evaluating improvements to Trace-Based Root Cause Localization and more accurate root cause localization. More information on the Ochiai method can be found in 7.4.

Though our initial evaluation of Trace-Based Root Cause Localization performed very well, beating out alternative approaches, there remain some concerns about the feasibility of storing a large number of traces as the number of requests scales, as outlined in Section 4.2. We therefore must attempt to use Trace-Based Root Cause

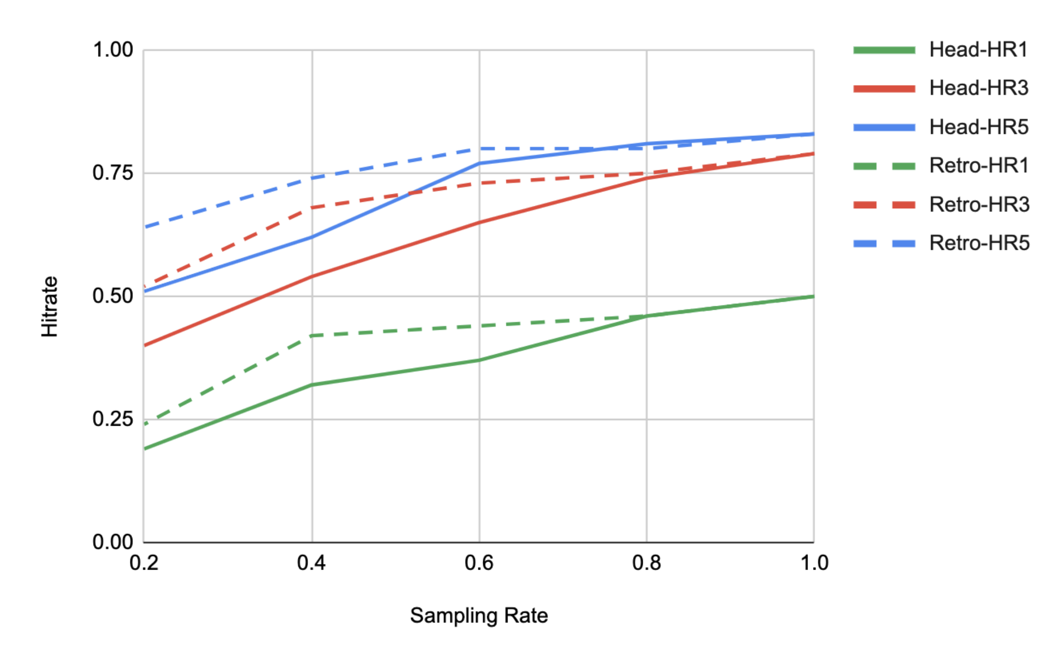

Figure 6.1: Heimdall Prediction Accuracy Under Sampling

Localization with a smaller sampling of the available traces, and determine whether it still performs well under our evaluation. We test this both under the random head-based sampling found in Jaeger [25], as well as under the retroactive sampling method [27].

In Figure 6.1, we observe the dropoff in each of the HR@K hitrate evaluation metrics as the percentage of traces sampled decreases. We test this under both approaches to sampling, with the solid lines representing head-based sampling, and the dashed lines representing retroactive sampling with the same sampling rate (representing the same constraints on trace storage). While we observe a dropoff under both methods, we see that the retroactive sampling method exhibits a slightly less severe dropoff is prediction accuracy as sampling rate decreases compared to the head-based sampling, indicating that this more targeted method of selecting which traces to keep is beneficial when dealing with root cause localization. These advantages of retroactive sampling are especially pronounced at lower sampling rates, with improvements of between 5 and 15% when the sampling rate is below 50%.

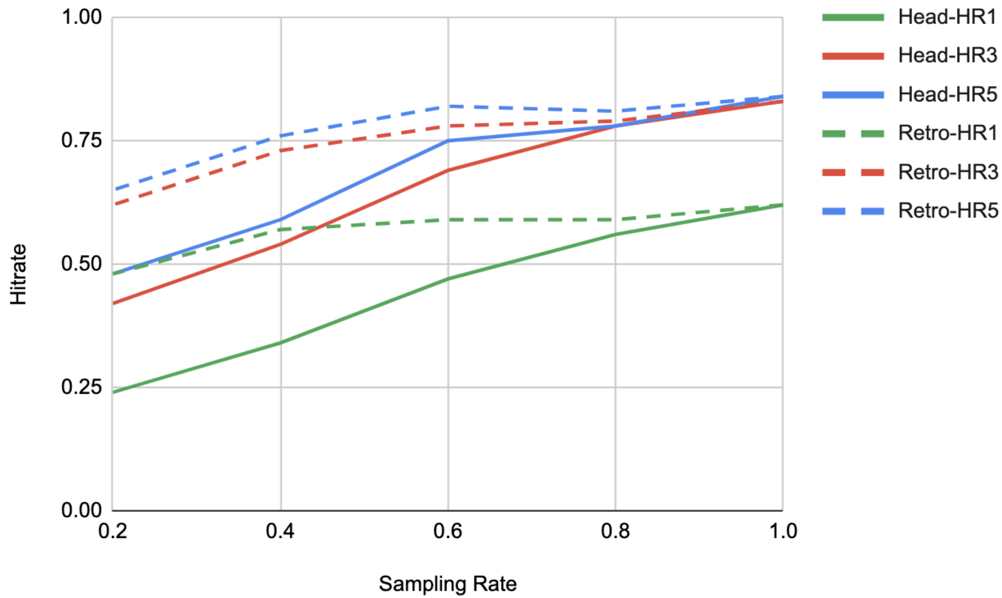

Figure 6.2: Ochiai Prediction Accuracy Under Sampling

Since the Ochiai localization method performed very well in the 100% sampling case, we also can observe how it performs under the two sampling methodologies. Figure 6.2 models the same dropoff in HR@k metrics for the Ochiai method, and once again we observe that this method seems to outperform the "suspiciousness ratio" method. Additionally, we observe that the Ochiai method performs exceptionally well at low sampling rates under retroactive sampling. In fact, at a sampling rate of 0.2 and using retroactive sampling, the Ochiai method scores at $HR@1 = 0.48$, $HR@3 = 0.62$, $HR@5 = 0.65$, and $MRR = 0.57$, still outperforming both the MonitorRank and Microscope projects while needed to store just 20% of the traces, significantly reducing the storage overhead.

The improved prediction accuracy of retroactive sampling of traces compared to head-based sampling at is in line with our thoughts in Section 4.2. Since head-based sampling randomly samples requests regardless of whether they are flagged as problematic, the chance of keeping the problematic traces decreases with the sampling rate This makes it difficult for Heimdall to gain any insights into what components

are associated with the problem, as the available set of problematic traces is small and incomplete. This becomes especially pronounced the more the system scales, as both the percentage of traces sampled and the threshold for problematic traces decreases, both of which reduce the chance that enough problematic traces are included in the sampled set.

Under retroactive sampling, the sampling method ensures that a full picture of the problematic traces is kept, making it easier for Heimdall to discern which components are strongly associated with the problem. Even under lower sampling rates, all the problematic traces remain in the sampled set, making the dropoff less severe. However, the fault localization techniques also rely on the healthy traces, which are still sampled at random under retroactive sampling. In fact, since all problematic traces are included in the sample set at a higher rate than the sampling rate, the healthy traces are included at a rate even lower than the sampling rate. Since the Trace-Based Root Cause Localization metric also considers both the executed and non-executed healthy traces when calculating its metric, we do still expect and observe a dropoff in prediction accuracy as the sampling rate, and therefore the number of collected healthy traces, decreases. The Ochiai method, on the other hand, considers the executed healthy traces, a^s_{ep}, but not the non-executed healthy traces, a^s_{np}. Since a large number of the traces that get filtered out by sampling fall into the a^s_{np} category, this may explain why the Heimdall metric, which uses this count, exhibits greater dropoff under sampling than the Ochiai measure, and this gap is even more pronounced under retroactive sampling. However, more in-depth theoretical analysis is needed to concretely evaluate the varying mathematical effects of sampling on each of the individual fault localization metrics.

Chapter 7

Related Work

7.1 Distributed Traces

Many modern-day applications are constructed as large-scale, distributed systems, composed of small interconnected microservices each deployed on individual machines, often in physically separate locations. As the number of microservices scales, the system becomes increasingly complex, with large numbers of system calls, numerous applications and servers to record, and dynamic graph topologies [6]. This makes it increasingly difficult to monitor the system and track issues that occur.

Given this challenge, a new approach is needed to track requests end-to-end through the distributed system, known as distributed tracing. Distributed tracing provides end-to-end information at the request level, detailing how a request was handled and serviced by the various components of the distributed system. In a large system, even a single request leads to numerous service calls across different machines and services, so a framework that links these calls is useful for gaining a broader view of how the system is running at a high level while still keeping the granularity at the service level as well.

Though adoption of distributed tracing for software fault localization has previ-

ously been fairly limited [6] due to some of the sampling challenges outlined in Section 4.2, distributed traces have been utilized to assist a variety of tasks. Prior works have identified distributed traces as especially helpful for proactive measures, such modeling workloads for resource and capacity planning to avoid future resource bottlenecks [27] [6].

In contrast with traditional logging, in which information about the executed code is recording locally on each machine, distributed tracing enables the ability to connect the information from different system components into a singular directed acyclic graph (DAG) for each request [4]. Such a graph also can provide an ordering of the services via a topological sort, diagramming which services are upstream or downstream relative to others. This ability to link the services under a single request ID is critical for many applications of distributed systems, including Trace-Based Root Cause Localization. Many distributed tracing frameworks, such as Jaeger [25], also provide an interface to visualize the trace, which can help the user understand the nature of the interconnected services in the application and the relationships between them, specifically service calls between adjacent services and upstream/downstream relationships [6].

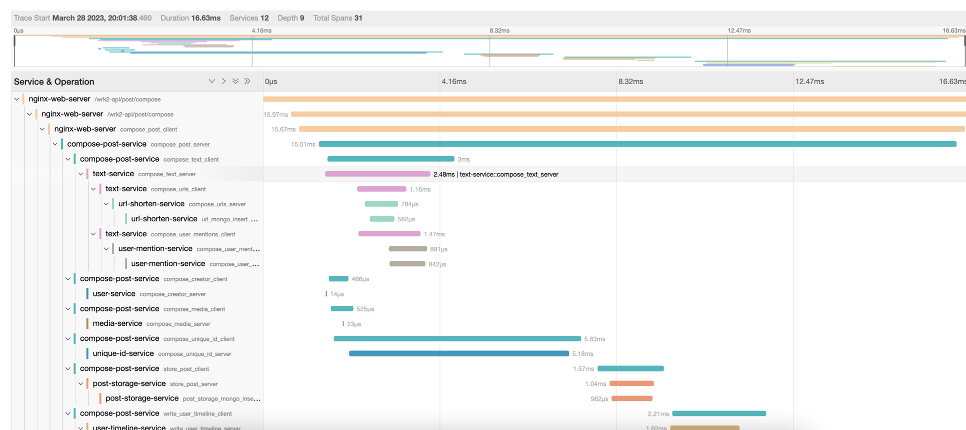

Figure 7.1: Sample Visualization of a Jaeger Trace from DeathStarBench

7.2 Retroactive Sampling

When collecting distributed traces, the tracing framework is capable of producing a massive number of traces, since logs and metrics can be produced at each stage of the system for each request, leading to a mass of traces that is impractical to store. As a result, many distributed tracing frameworks, such as Jaeger, randomly sample a subset of requests to trace and store, usually using head-based sampling. This style of sampling is an issue when trying to troubleshoot an issue as an engineer, as the edge-case issue that caused the problem may not be present in the traces at all. Similarly, when attempting to diagnose the root causes of issues using Heimdall, it is imperative that Heimdall has access to the faulty traces, as well as an adequate number of healthy traces, to accurately localize the issue. To ensure that the relevant traces are captured, Heimdall can be paired with a distributed tracing framework that uses retroactive sampling, as proposed by Hindsight [27]. Hindsight records all data locally for a short amount of time, so that any request that is reported to show poor performance upon completion can then have its trace collected for analysis. This model of tracing allows the tracing framework to record any edge-case performance issues that occur in a small subset of requests, and may not have otherwise been collected. In conjunction with Heimdall, retroactive tracing can be utilized to quickly localize and diagnose small, edge-case issues that might have otherwise gone unnoticed.

7.3 DeathStarBench

DeathStarBench [9] is an open-source benchmark suite for cloud-based applications with microservice architecture, providing multiple end-to-end services that can be used to test and benchmark microservice-related tools and systems. DeathStarBench offers 5 total services; for this project we used the social network service as our benchmark and testbed for latency injection and performance localization tests with

Heimdall. A diagram of the application structure can be found in Figure 7.2. DeathStarBench can be deployed as containerized services with Docker, and comes pre-instrumented with Jaeger tracepoints in each microservice.

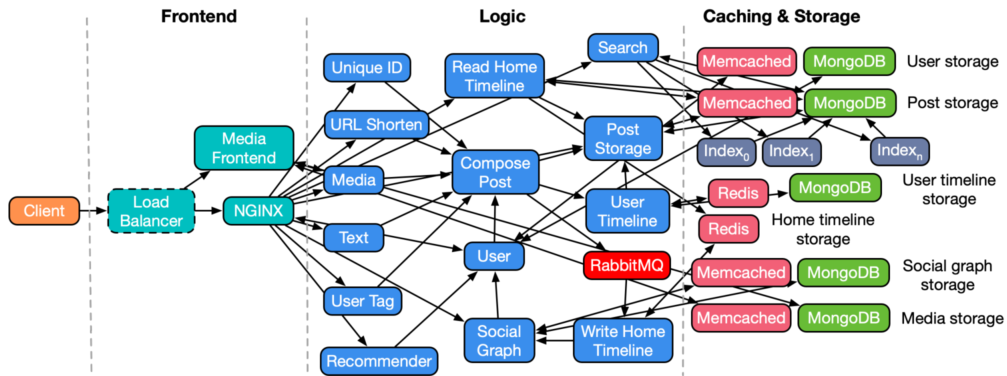

Figure 7.2: Architecture of DeathStarBench Social Network

This small microservice system benchmark simulated a social network where users can post to their timeline, send follow requests, and view the posts of others, each of which has their own type of request that can be sent. For our experiments, we sent compose-post requests, which simulate a user posting to their timeline. As part of servicing this request, the request passes through individual microservices that perform various small tasks, such as writing the post to a database or notifying other users that were mentioned in the post.

7.4 Fault Localization

There are a number of prior works that explore the idea of software fault localization. Early work attempts to create a generalized model that for localizing a single error in a single program [21] [1], and design automatic tools which can diagnose faults in a program [2] [12]. Various different fault localization or ”suspiciousness” metrics have been proposed [22] [2] [14] [26], each of which has been demonstrated to be useful in specific use cases, although no single metric performs best in all cases. Practical

work in software fault localization evaluate the usefulness of these metrics in various real world cases[24] [14]. Most of these real-world applications of fault localization techniques focus on a single program, where parts of the program are executed or not executed by some sort of if-else logic. By contrast, Heimdall applies these techniques to the distributed system, where code may be executed across different machines, and decisions over what code chunks are executed are not just due to if-else logic, but other features of the system such as random load balancing.

We applied several of these metrics to our trace data as a way to benchmark the performance of our new suspicious ratios against previous fault localization tools. Notably we found the Ochiai fault localization method [22] performed very well in our experiments, scoring the highest among the fault localization techniques across all evaluation metrics. The Ochiai factor, originally developed in a biological study of Japanese fish, is effectively a coefficient of similarity between two groups or sets. When applied to the fault localization use case, the Ochiai coefficient is defined as as:

$$\frac{a^s_{ef}}{\sqrt{(a^s_{ef} + a^s_{nf})(a^s_{ef} + a^s_{ep})}}$$

In this case, $(a^s_{ef} + a^s_{nf})$ is the set of triggered traces, and $(a^s_{ef} + a^s_{ep})$ is the size of the executed cases for some component a. In the numerator, we have the size of the overlap between these two cases, a^s_{ef}. Thus, the above ratio expressed the similarity between the triggered and executed groups, i.e. the level of association between a system component and problematic, poorly performing traces.

7.5 Benchmarks

In Section 6, we evaluated the accuracy of the Heimdall rankings against various benchmarks. Some of these benchmarks were the fault localization metrics defined in Section 3.4; however we also compared against the predictions of three existing

tools for fault or performance anomaly localization in microservice systems, namely MicroHECL [17], MonitorRank[15], and Microscope [16]. Each of these systems takes their own slightly different methodological approach to the localization problem, and used different microservice systems for their experiments and evaluation.

MicroHECL [17] is a root cause localization tool developed by Alibaba that localizes the root cause for not just performance problems, but also other kinds of issues commonly faced by the Alibaba system, such as outages. MicroHECL ingests information about service calls, service relationships, and dependencies in the system, and uses machine learning techniques to construct a dynamic service call graph that is analyzed to determine how anomalies propagate through services and where they originated. MicroHECL was tested on and deployed into production for the Alibaba system, a massive production system with over 30,000 total microservices that comprise over 300 subsystems.

MonitorRank [15] takes a similar approach to MicroHECL, generating a graph based on service calls between system components. They also used logs and metrics from within each service to assist in the model building and fault localization predictions. Like MicroHECL, MonitorRank also focused on both performance and reliability anomalies, and was tested on production-scale system data at LinkedIn, with about 400 unique services deployed across thousands of nodes.

Microscope [16] also uses call data to generate a graph, algorithmically extracting causal relationships between services. Unlike the other two, Microscope focuses solely on performance with respect to latency rather than a wider range of issues, just like Heimdall. Microscope was evaluated in a non-production environment, tested on the sock-shop benchmark system, comprised of 36 service instances.

Each of these approaches takes a slightly different approach from one another, but they all have some clear similarities that make them distinct from Heimdall. Most notably, these tools all ingest information about service calls between microservices,

and constructs a graph that represents relationships between nodes. None of these tools utilize distributed traces for their analysis. MicroHECL notes the challenges of using traces, namely the large overhead needed to store traces at scale. However, when combined with retroactive sampling, Heimdall is able to avoid this concern. By avoiding the graph based approach, Heimdall is also able to avoid any time or space overhead that comes with generating this service call graph, which can become highly complicated and large when the number of services scales up, as is the case in large production systems. MonitorRank mentions the difficulty of service call graph generation and analyzing issue propagation, as the root cause localizer needs to analyze each edge of the call graph, which could theoretically scale quadratically with the number of services.

7.6 Distributed System Outage Detection

Since distributed systems operate at great scale, the quantity and diversity of potential issues is high. Some prior work has attempted to classify these issues based on the nature of their root causes and symptoms [30] [29]. These papers utilize similar failure injection and fault localization techniques, and we draw from their notation and evaluation frameworks.

However, all these analyses focus primarily on faults or outages, in which the system shuts down or fails to complete requests. This is different from the Heimdall use case, in which affected "bad" traces are classified as traces that exhibit poor performance (usually high latency), but do not fail entirely. While the methodologies may overlap, Heimdall deals with a much smaller set of potential issue types, and problematic traces are often somewhat less noticeable.

7.7 Bayes Factor

The Bayes Factor is a measure proposed by Sir Harold Jeffreys [13] as a tool to contrast the predictive performance of two competing models or hypotheses [19], measuring how much the data supports one statistical model over another as a ratio. The Bayes factor has found application across a wide range of disciplines in both the natural sciences and social sciences, and are considered "the primary tool used in Bayesian inference for hypothesis testing and model selection." [5] [8]

To calculate the Bayes Factor, we consider two statistical models or hypothesis. Under the traditional framework, these are the null hypothesis H_0, signifying the absence of an effect, and the alternative hypothesis H_1, signifying the presence of an effect. [19]. The Bayes factor also requires some already observed data, notated as d. Then, we can express our Bayes factor as:

$$\frac{p(d|H_1)}{p(d|H_0)}$$

The Bayes factor can be interpreted as the ratio of the average predictive accuracies of the two models, or the ratio of the likelihoods that the observed data would be generated under that model. Thus, a Bayes factor of 3 indicates that the data is 3 times more likely to occur under the model of the alternative hypothesis H_1 than the null hypothesis H_0, while a Bayes factor of 0.125, or 1/8, indicates that the data is 8 times more likely to occur under H_0 than H_1.

Chapter 8

Conclusion and Future Work

Trace-Based Root Cause Localization scores highly in measures of its predictive accuracy, outperforming many existing fault localization tools that opt to not leverage distributed traces. Even existing fault localization metrics are found to perform very well when coupled with distributed tracing, underlining the usefulness of distributed traces and Trace-Based Root Cause Localization in the field of performance regression localization. We also demonstrated the advantages of using retroactive sampling as part of the localization process instead of head-based sampling, providing evidence in support of the belief that use of retroactive sampling is advantageous when localizing problems in a distributed system and addressing concerns in prior work about the feasibility of using tracing at scale for issue localization.

More work is needed to test the limits and shortcomings of Trace-Based Root Cause Localization under a variety of issue types and at varying levels of scale. Use of Trace-Based Root Cause Localization in a production level system, where the number of services and the diversity of issues is high and the sampling rate must be relatively low, can provide further insights into the viability of using distributed tracing for fault localization at scale.

Additional adjustments, such as incorporating ideas from the Ochiai method or

other techniques, might also be made to the fault localization metric to improve predictions either in the general case or when applied to specific systems. More complex analytical work should also to done to determine *why* some methods outperform others, and what qualities of the system or recorded data can be leveraged to efficiently identify the most accurate prediction method for root cause localization. Being able to identify the mathematical foundations that cause a distributed tracing approach to outperform a service call graph approach, or understanding why Ochiai performs better than other localization methods, can allow greater understanding of the problem space and more efficient and guided future work.

Bibliography

[1] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. An evaluation of similarity coefficients for software fault localization. In *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*, pages 39–46. IEEE, 2006.

[2] Rui Abreu, Alberto González, Peter Zoeteweij, and Arjan JC van Gemund. Automatic software fault localization using generic program invariants. In *Proceedings of the 2008 ACM symposium on Applied computing*, pages 712–717, 2008.

[3] Akamai. Akamai online retail performance report: Milliseconds are critical, 2017. URL `https://www.akamai.com/newsroom/press-release/akamai-releases-spring-2017-state-of-online-retail-performance-report`.

[4] Andre Bento, Jaime Correia, Ricardo Filipe, Filipe Araujo, and Jorge Cardoso. Automated analysis of distributed tracing: Challenges and research directions. *Journal of Grid Computing*, 19:1–15, 2021.

[5] James Berger and Luis Pericchi. Bayes factors. *Wiley StatsRef: statistics reference online*, pages 1–14, 2014.

[6] 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*, 2023.

[7] Dynatrace. 2022 dynatrace cio report: Retail. `https://www.dynatrace.com/info/reports/retail-cio-report/`, 2022.

[8] Alexander Etz and Eric-Jan Wagenmakers. Jbs haldane's contribution to the bayes factor hypothesis test. *Statistical Science*, pages 313–329, 2017.

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

[10] Supriyo Ghosh, Manish Shetty, Chetan Bansal, and Suman Nath. How to fight production incidents? an empirical study on a large-scale cloud service. In *Proceedings of the 13th Symposium on Cloud Computing*, pages 126–141, 2022.

[11] Olumuyiwa Ibidunmoye, Francisco Hernández-Rodriguez, and Erik Elmroth. Performance anomaly detection and bottleneck identification. *ACM Computing Surveys (CSUR)*, 48(1):1–35, 2015.

[12] Tom Janssen, Rui Abreu, and Arjan JC Van Gemund. Zoltar: A toolset for automatic fault localization. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 662–664. IEEE, 2009.

[13] Harold Jeffreys. *The theory of probability*. OuP Oxford, 1998.

[14] James A Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 273–282, 2005.

[15] 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.

[16] 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.

[17] Dewei Liu, Chuan He, Xin Peng, Fan Lin, Chenxi Zhang, Shengfang Gong, Ziang Li, Jiayu Ou, and Zheshun Wu. Microhecl: High-efficient root cause localization in large-scale microservice systems. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 338–347. IEEE, 2021.

[18] Haopeng Liu, Shan Lu, Madan Musuvathi, and Suman Nath. What bugs cause production cloud incidents? In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 155–162, 2019.

[19] Alexander Ly, Alexander Etz, Maarten Marsman, and Eric-Jan Wagenmakers. Replication bayes factors from evidence updating. *Behavior research methods*, 51:2498–2508, 2019.

[20] Linux User's Manual. *usleep(3)*, 2021.

[21] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. A model for spectra-based software diagnosis. *ACM Transactions on software engineering and methodology (TOSEM)*, 20(3):1–32, 2011.

[22] Akira Ochiai. Zoogeographic studies on the soleoid fishes found in japan and its neighbouring regions. *Bulletin of Japanese Society of Scientific Fisheries*, 22: 526–530, 1957.

[23] 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.

[24] Youcheng Sun, Hana Chockler, Xiaowei Huang, and Daniel Kroening. Explaining image classifiers using statistical fault localization. In *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part XXVIII*, pages 391–406. Springer, 2020.

[25] Uber. Jaeger. URL `https://www.jaegertracing.io/`.

[26] W Eric Wong, Yu Qi, Lei Zhao, and Kai-Yuan Cai. Effective fault localization using code coverage. In *31st Annual International Computer Software and Applications Conference (COMPSAC 2007)*, volume 1, pages 449–456. IEEE, 2007.

[27] Lei Zhang, Vaastav Anand, Zhiqiang Xie, Ymir Vigfusson, and Jonathan Mace. The benefit of hindsight: Tracing edge-cases in distributed systems. *arXiv preprint arXiv:2202.05769*, 2022.

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

[29] 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*, 47(2):243–260, 2018.

[30] 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.