2023

Benchmarking microservices: effects of tracing and service mesh

BOSTON UNIVERSITY

GRADUATE SCHOOL OF ARTS & SCIENCES

Thesis

BENCHMARKING MICROSERVICES: EFFECTS OF TRACING AND SERVICE MESH

by

VIVEK UNNIKRISHNAN

B.Tech., National Institute of Technology, Warangal, 2017

Submitted in partial fulfillment of the

requirements for the degree of

Master of Science

2023

Approved by

First Reader

Ioannis Liagouris, PhD
Assistant Professor of Computer Science

Second Reader

Vasiliki Kalavri, PhD
Assistant Professor of Computer Science

Third Reader

Orran Krieger, PhD
Professor of Electrical and Computer Engineering

Out of clutter, find Simplicity
From discord, find Harmony.
In the middle of difficulty lies Opportunity.
- Albert Einstein

Acknowledgments

First and foremost I am extremely grateful to my supervisors, Dr. Ioannis Liagouris and Prof. Vasiliki Kalavri for their invaluable advice, continuous support, and patience during my Master's study. Their immense knowledge and great experience have encouraged me throughout my academic research and daily life. I would like to thank all the members of the CASP Lab and the BU GRS CS course. It is their kind help and support that have made my study and life in the US a wonderful time. Finally, I would like to express my gratitude to my parents and my brother. Without their tremendous understanding and encouragement, it would be impossible for me to complete my study.

Vivek Unnikrishnan

CS Department

BENCHMARKING MICROSERVICES: EFFECTS OF TRACING AND SERVICE MESH

VIVEK UNNIKRISHNAN

ABSTRACT

Microservices have become the current standard in software architecture. As the number of microservices increases, there is an increased need for better visualization, debugging and configuration management. Developers currently adopt various tools to achieve the above functionalities two of which are tracing tools and service meshes. Despite the advantages, they bring to the table, the overhead they add is also significant. In this thesis, we try to understand these overheads in latency and throughput by conducting experiments on known benchmarks with different tracing tools and service meshes. We introduce a new tool called Unified Benchmark Runner (UBR) that allows easy benchmark setup, enabling a more systematic way to run multiple benchmark experiments under different scenarios. UBR supports Jaeger, TCP Dump, Istio, and three popular microservice benchhmarks, namely, Social Network, Hotel Reservation, and Online Boutique. Using UBR, we conduct experiments with all three benchmarks and report performance for different deployments and configurations.

Contents

List of Tables

List of Figures

List of Abbreviations

RPC	.....	Remote Procedural Call
TCP	.....	Transmission Control Protocol
UBR	.....	Unified Benchmark Runner

Chapter 1

Introduction

1.1 Microservices

Microservices have become a part of most production applications today. Unlike monolithic applications in which one application contains multiple functionalities, each microservice essentially has a single functionality or handles a single domain such as storage, authorization or monitoring. This makes the microservice architecture fault-tolerant, loosely coupled and easily scalable. Deployments are easier as well since only specific microservices need to be updated and this can be done easily with minimal downtime.

But with the increased number of services to manage few issues also come up: Mainly error detection and performance monitoring. As there are more services to handle, the points of failure also increase. Tracing and monitoring tools help track the requests that flow through microservices and understand the possible points of contention. Tools like these add overhead in terms of latency, throughput and overall performance despite giving advantages. We try to understand the overheads of these methods by running them on existing microservice benchmarks so as to make informed decisions about accepted overheads and required resources for microservices.

1.2 Tracing

In order to understand the effect of tracing on microservices a few popular tracing methods were selected to understand the overhead each method has.

1.2.1 TCP Dump

TCP Dump is not a tracing method in the traditional sense. But the packets obtained using TCP Dump (tcp, 2022) can be used to construct a trace. It is also a good starting point to understand what would be the overhead on the microservice system if every packet is tracked and stored for analysis. In order to trace the microservice pods, an open-source tool called Ksniff (Ksn, 2022) was used to launch pods running TCP Dump which will listen to the packets of a specified pod and dump the packets to an output file. These packets would then be analysed using a Python script to create a CSV file containing information about the source of the packet, the destination of the packet, the unique packet sequence number and the headers generated in the packet. The above information can then be utilized to construct a trace of all the API calls during the benchmark. The downside of TCP Dump was that the traces were incomplete unless a unique context header was passed along with every request. This problem can be tackled using service meshes or making sure every microservice follows the Open tracing guidelines for better traceability.

1.2.2 Jaeger

Jaeger (Jae, 2022) is a popular open-source distributed tracing system inspired by Google's Dapper paper. It can be used to trace microservices for root cause analysis and performance optimization. Jaeger works by collecting trace information from various microservices. In Jaeger, traces are made up of multiple spans. Developers can add spans to their code to understand which parts of their code take the most

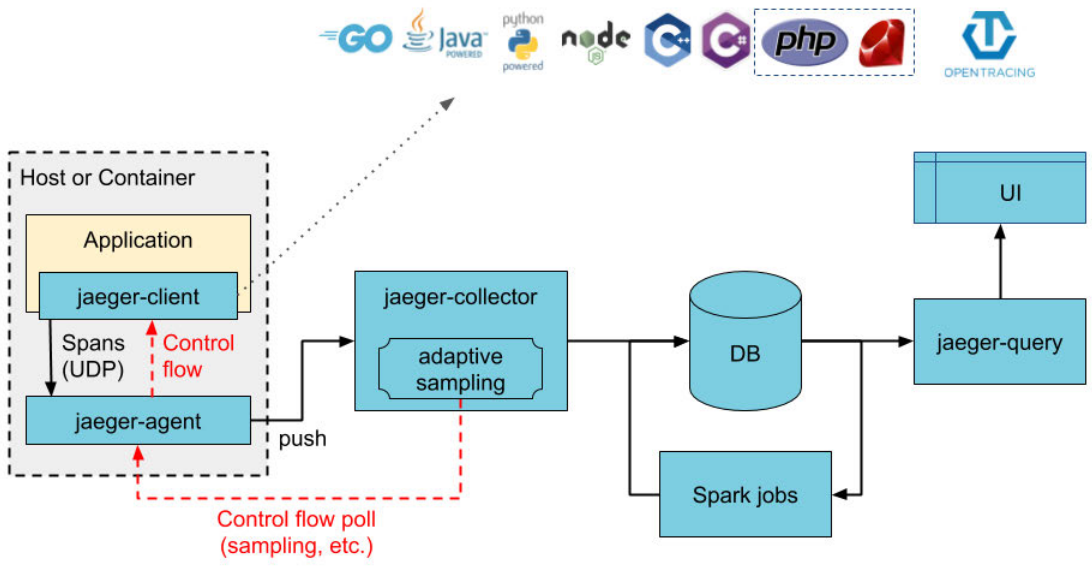


Figure 1·1: Jaeger Architecture (Jae, 2022)

time. Jaeger introduces overhead to the system since the information needs to be collected and sent to the Jaeger client. This overhead depends on the sampling rate, and the number of spans collected as part of the tracing.

1.3 Service Mesh

A service mesh is a dedicated infrastructure layer that developers can deploy. Once deployed it takes care of multiple functions such as tracing, monitoring, rate limiting and reducing downtime. Due to the huge number of advantages a service mesh provides, the downsides of latency and throughput overhead are often ignored. Further, the resources required to run a cluster with service mesh enabled also increase as the rate of requests increases. The two popular service mesh options for Kubernetes environments are Linkerd (Lin, 2022) and Istio (Ist, 2022). In our benchmarks, we use Istio since it is more popular and has more features than Linkerd.

1.3.1 Istio

Istio is an open-source service mesh which provides a uniform way to manage deployed services. It is divided into two parts: The control plane and the data plane. The control plane deals with the configuration of Istio and dynamically provisions the sidecar proxy servers when the configuration changes. The main component of the control plane is Istiod which consists of various sub-components that deal with configuration management, service discovery and monitoring. The data plane handles the communication between the different microservices. The main component of the data plane is the Envoy proxy, commonly referred to as sidecars. The Envoy proxy is the heart of Istio and it helps form a secure service mesh with functions like route discovery, Layer-7 routing, rate limiting and filters. Figure 1·2 shows the architecture of Istio with the various components discussed above.

1.4 Thesis Contributions

As part of the thesis, we have developed a new tool called the Unified Benchmark Runner (UBR) which helps generalize the benchmark execution process as well as set standards and templates for any future benchmark that needs to be executed. We use UBR to run experiments on three different benchmarks: social network, hotel reservation and online boutique. The experiments involve understanding the latency, throughput and resource overhead when the benchmarks are run after installing one of the three different tools: TCP Dump which is a packet analyzer, Jaeger which is a tracing tool and Istio which is a service mesh. Based on the experiments, here are the key findings:

- TCP Dump is fairly lightweight, adding throughput overhead between 4%-15% and latency overhead between 5%-33% to the microservices, depending on the benchmark it is run on.

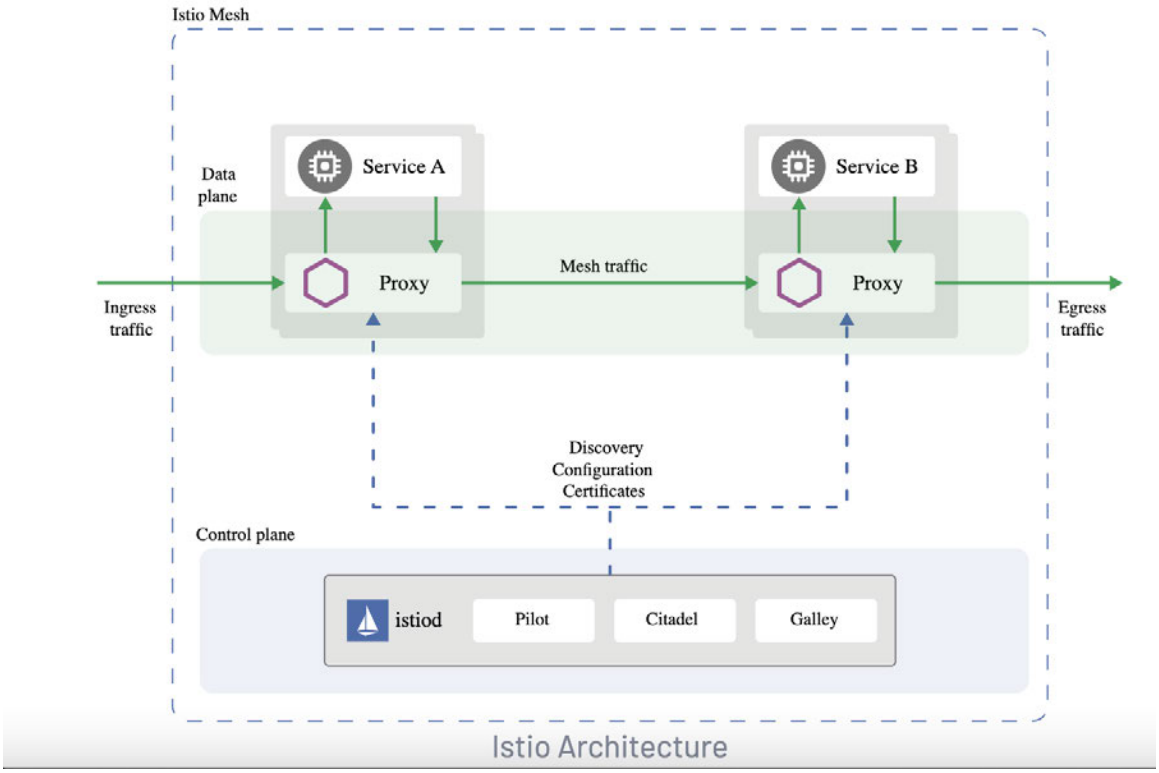


Figure 1·2: Istio Architecture (Ist, 2022)

- Jaeger's overhead heavily depends on the number of microservices and trace sampling, i.e., the amount of data sent to the agent.
- Istio has throughput overhead between 15%-88% and latency overhead between 16%-400% depending on the benchmark it is run on.

Chapter 2

Related Work

This chapter explores previous research in causal profiling and microservices. We will first review current profiling methods after which we will go into papers that have done work on microservice performance improvement and end-to-end tracing of requests within microservices.

2.1 Understanding Microservice Performance

Unlike monolithic applications, microservices are separate applications doing specific functions. They can be deployed anywhere. A variety of factors determine a microservice performance including but not limited to the amount of memory and CPU assigned, the placement of the services and the number of replicas of the service. It is crucial to understand the factors affecting the performance of a microservice. In the below sections, we will go through a few works that will help us understand how the microservice works and how we can identify these factors for better optimizations.

2.1.1 Distributed Tracing

Tracing requests and processes within systems is a crucial task, especially in critical distributed systems where there are multiple points of failure. But it is not a one size fits all solution. As mentioned in the survey (Sambasivan et al., 2014) there are a few design axes that need to be considered while designing a tracing system. This includes: what causal relationships need to be preserved, how they are tracked, how

much overhead is acceptable and what kind of visualization is needed. The current tracing product that is used depends on the problem that we are trying to solve. For example, if anomalies in the distributed system need to be surfaced and visualized a tool like Magpie would be more suitable whereas profiling the distributed system could be done better by a tool like Dapper. Most tools use one of the 3 methods for end-to-end tracing. The first one is metadata propagation which is commonly used especially in the case of microservices. A context or an ID is propagated across requests to ensure that we can connect all interactions required for visualizing the system (Dapper based systems like Jaeger). The major downside of this approach is the overhead it adds to the request both in terms of size and latency. The second is the schema-based approach where developers write temporal join schemas within the code to establish causal relationships (ETE and Magpie). The downside of this approach is that it is not scalable since developers need to make sure the schema relationships are always maintained. The third approach is using grey-box inference. This approach tries to solve the problem by understanding relationships from logs, variables and timestamps produced by the distributed system. The pros include non-modification of code and very negligible overhead but the major downside is the low accuracy of such methods as it depends on information generated by the microservices which might not be well formatted or designed.

The survey delves deeper into which causal relationships need to be preserved. Most distributed tracing infrastructures resort to preserving the happens-before relation. But not all slices of the relation are relevant to tracing and tracking different slices is used for achieving different results within the tracing. Mainly they are divided into two classes: Intra-request slices for all happens-before relations within the same request and inter-request slices for relations between multiple requests. Intra-request slices help to understand and profile individual requests and the time spent for

each microservice in the request. Intra-request slices can be further categorized into submitter-preserving and trigger-preserving slices. Submitter-preserving slices show causality between the original submitter of a request and the work done to process the request in every component of the system. This is useful for resource attribution since we need to know which resource has done what work. Trigger preserving on the other hand is attributed to the trigger of the process, not the submitter. This helps in understanding anomalies in the request and diagnosing problems with performance. Inter-request slices are used to understand problems with resource contention when two requests utilize the same resources like databases or memory.

One important aspect of the survey is regarding overheads produced by these distributed tracing systems. As mentioned Dapper, one of the most used distributed tracing services introduces 1.5% throughput overhead and 16% latency overhead to every request at a 100% sampling rate. The overheads can be reduced by adjusting the sampling percentage and strategy.

The survey serves as a good starting point to understand what are the key points to keep in mind while designing a distributed tracing system. It also sheds light on the performance overhead aspect of tracing and what to keep in mind while trying to reduce this overhead.

2.1.2 3MileBeach

While looking into tracing solutions that are better than the current market-leading ones such as Jaeger or Zipkin, we came across 3MileBeach (Zhang et al., 2021). The application uses the message serialization library layer to provide fine-grained tracing and fault injection capabilities transparently by using message headers as a temporary data store. This helps 3MileBeach overcome one of the main disadvantages of Dapper and dapper-based solutions such as Jaeger: lost or delayed traces. Since the trace is only propagated along the request flows there is a possibility of a loss of traces

which can lead to incomplete subgraphs while parsing requests. Also due to the fewer interaction network calls to the agent, the overhead is also less compared to Jaeger in which traces are sent to the agent every time. The downside of this approach is that the message header can grow large in size. Since 3MileBeach uses the libraries used for serialization and deserialization for propagating context and trace metadata, every message that is received contains the complete causal history of the service calls carried by the trace events. This history is further used for fault injection and identifying errors between microservices including TOCTOU (Time to check to Time to use) bugs which are very difficult to identify.

Connecting multiple requests was crucial to creating a proper tool for tracing and analysis and 3MileBeach solved this issue using request headers and storing request identifiers in memory. The trade-off, in this case, was increased size of the packet leading to increased size in packet compared to Jaeger which has smaller packets but a higher number of packet transfers to a centralized Jaeger client leading to overhead. The Jaeger and 3MileBeach comparison results were used to compare the Jaeger overheads we were observing with the same benchmark and setup.

2.1.3 CRISP

The idea of analysis of results produced by tracing tools such as Jaeger was also important to be understood. Considering we can build a tool that can profile a distributed system, it would be useful when it can be analyzed effectively. This is especially true in the cases where generated traces are typically large and their analysis is computationally expensive. CRISP(Zhang et al., 2022) is a tool to perform critical path analysis using a large number of traces that are collected using Jaeger. It provides a top-down CPA analysis, bottom-up CPA analysis and anomaly detection to alert potential problems. Tested in Uber over three months on 40K endpoints while processing 200GB of traces, CRISP is suitable for dealing with a large number of

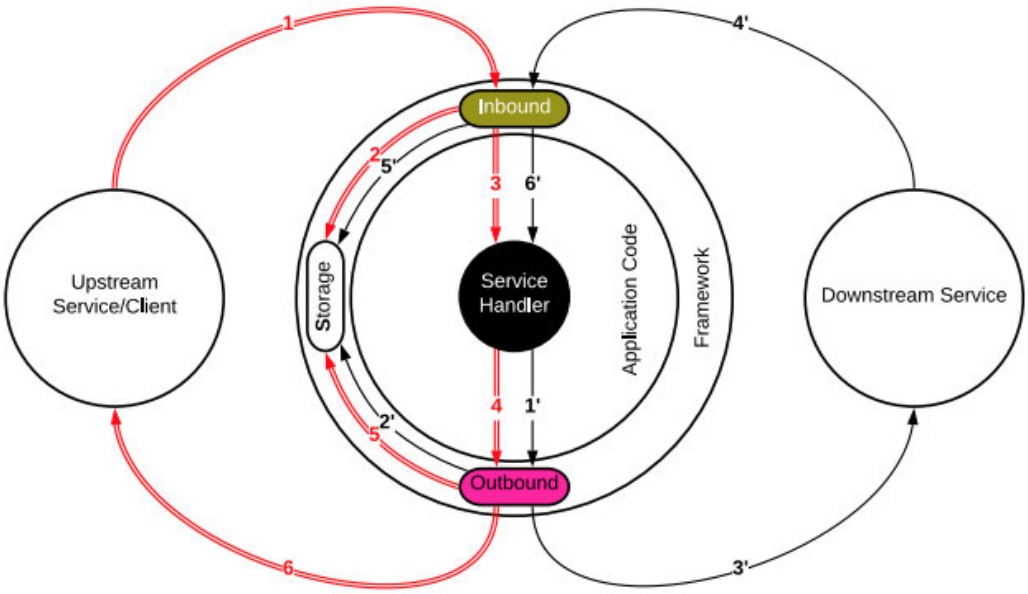


Figure 2·1: The cell model used for 3MileBeach Architecture (Zhang et al., 2021)

Jaeger traces. There are three main features offered by CRISP: Top-down analysis which is represented as a flame graph and a heat map and Bottom-up analysis where all endpoints are analyzed to bubble up the most used interior APIs so as to improve them. These are represented as histograms. CRISP also has an anomaly detection function using a machine learning model to know how much the incoming trace deviates from normal execution behaviour. For this, the critical path is encoded into Service Critical path vectors to learn the normal execution pattern.

The main idea of the CRISP paper is about how critical path analysis can be done effectively using Jaeger traces. If we can get a similar structure of traces from newly developed tracing methods we can likely use CRISP to generate critical paths and use them for analysis similar to the ones done above.

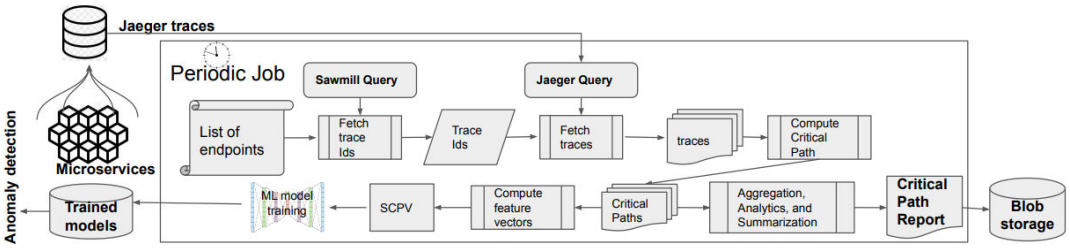


Figure 2·2: Architecture of the CRISP tool (Zhang et al., 2022)

2.1.4 Stitch

One of the areas we are looking into is whether we can leverage the use of logs generated by Openshift to profile the microservices. We came across this tool called Stitch(Zhao et al., 2016), a non-intrusive tool designed to help debug code using unstructured logs provided by the developers. The tool is based on the Flow Reconstruction principle which states that programmers will log events that can be used to reconstruct the execution flow. The method focuses on objects in particular and their relationships and interactions. This is used to create a program activity graph where we can focus on specific objects and find the lines of code that take the most time. The output of the algorithm is a Directed Acyclic graph known as the System Stack Structure graph. The major downside of Stitch is that it heavily relies on the logs the application produces and this might not always be accurate unless the application follows certain programming design practices. The other aspect is that Stitch was designed for monolithic applications and not microservices. Microservices can be mapped to objects but the distributed aspect of microservices would further reduce the accuracy of Stitch as multiple logs can appear at the same timestamp across different microservices.

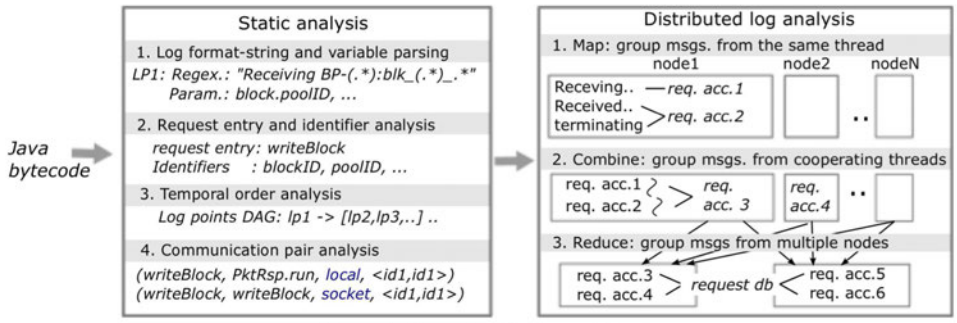


Figure 5: Overall architecture of *Iprof*

Figure 2·3: High-level overview of IProf (Zhao et al., 2014)

2.1.5 IProf

IProf (Zhao et al., 2014) is a non-intrusive profiling tool that helps analyse and debug distributed systems' performance. The main idea is to extract per request information about performance from logs that are generated by the application. The main advantages of IProf are that it is non-intrusive, scalable, allows historical analysis and is more robust to data loss. Iprof has 2 main steps that are run to analyze the logs as shown in Figure 2·3:

- Static analysis of the logs to map each log message to a specific request and get the order of log messages mapped to the request
- Log Analysis using Map reduce and the summary information obtained from static analysis

The main idea here is to analyze Java byte code by parsing log statements produced to get key request identifiers and create a directed acyclic graph. The identifiers and the graph can then be used in the Log analysis stage to identify key communication insights between different processes happening in the cluster.

IProf has been tested on a 200 node EC2 cluster using Hibench benchmark for HDFS and Yarn, and YCSB for Cassandra and HBase. The static analysis step takes

2 minutes to generate the result after which the log analysis step generates the report with an 88% accuracy of attributing the log messages to the right request.

IProf helps understand how log analysis can bring out fairly accurate analysis over distributed systems. The use of request identifiers which are utilized by major logging libraries including python logging and SLJ4 can be utilized to build a tracing system with lower overhead. The only downside of IProf is that it is heavily dependent on Java Byte code. Microservices, in general, are written in multiple programming languages and thus having a solution that utilizes the intricacies of a specific programming language would restrict the solution in many ways. The flow of analysis could be utilized as we design a system for profiling microservices running on Openshift.

2.2 Optimize Microservice Performance

Once we understand the factors affecting the microservice performance, optimizing the factors is the next step. The below sections go through a few tools that analyse data produced by various tools to optimize microservice performance.

2.2.1 SHOWAR

SHOWAR (Baarzi and Kesidis, 2021) addresses the challenges in deploying microservices: mainly finding the optimal amount of resources (i.e size) and the number of instances (i.e replicas). SHOWAR takes care of 3 main aspects: cluster resource allocation, scaling and placement. It is designed for both vertical and horizontal scaling: Vertical autoscaling uses variance in historical resource usage to find optimal resource sizing of each microservice. This mainly deals with the CPU and memory of the microservice. Horizontal autoscaling uses the runq latency from Linux kernel library eBPF to make accurate scaling decisions using a proportional integral derivative (PID) controller

Table 2.1: Overview of tools to understand microservices

Tool	Intrusive	Source	Language	Features	Visualization
Stitch	No	Logs	Any	• Easier Error detection • Debugging tool	System Slack Graph
IProf	No	Logs and Byte Code	Java	• DAG Creation • RIC Set creation • Temporal flow analysis	Website for request analysis
3MileBeach	No	Requests	Any	• Fault Injection • Distributed Tracing	None
CRISP	Yes	Jaeger Traces	Any	• Critical Path Analysis • Anomaly Detection	Flame Graph

The tool is implemented in Golang as modules for Kubernetes. The package includes monitoring agents, vertical and horizontal autoscaler and the affinity rules generator. The entire overhead is minimal compared to the default Kubernetes autoscaler since it does not share resources with the other pods by making sure it is deployed on the controller pod. It helps understand the CPU and memory contentions that affect performance. These contentions can be further used to explain the difference in performance seen when tracing or service meshes have been added to the microservice.

2.2.2 MeshInsight

Service meshes are becoming an essential part of microservices. It provides an easy and flexible way to communicate with multiple microservices with the added advantage of enabling configuration, load balancing, rate limiting, tracing or any other features uniformly across various microservices.

In service meshes, all traffic traverses through software proxies called sidecars. This increases request latency and leads to the consumption of more resources. These overheads can degrade the user experience. The paper (Zhu et al., 2022) talks about how these overheads occur and what can be done to solve the question of what optimizations are needed to maximize resource usage while using a service mesh. Using the tool Meshinsight mentioned in the paper, we can estimate the end-to-end overhead as well as the impact of optimizations on a cluster of microservices. Running Meshinsight on 2 popular benchmark applications showed that depending on the configuration and the optimizations done the request latency increases up to 185% and CPU usage increases up to 92%.

Every service mesh has 2 main parts: the data plane and the control plane. The control plane deals with the discovery of pods, metrics gathering and central configuration of the applications produced by the data plane. The data plane injects

sidecars into pods. Once the microservices is part of a service mesh there are 3 major connections that happen in the microservice when it receives a request: two of these communications happen between the microservice and the sidecar and one happens between the sidecars. The data plane overhead is the most important since it is on the critical path of user experience. Meshinsight tries various configurations of sidecar protocols and filters (provided by the Service mesh) to generate latency and CPU profiles These profiles are then used along with the annotated call graph to make predictions of performance. Depending on the use case, one can select the configuration required for the application to ensure minimum overhead and maximum use of the resources provided.

Table 2.2: Overview of tools to optimize microservice performance

Tool	Parameters Optimized	Language	Features
SHOWAR	• CPU • Memory • Replica Count • Placement	GoLang	• Vertical Scaling using history • Horizontal scaling using eBPF • Service placement analysis
Meshinsight	• CPU • Memory	Python and Perl	• Overhead analysis • Performance prediction • Best configuration generation

Chapter 3

Methodology

In this chapter, we will go through the methodology of testing the overheads that different tracing and service mesh add to the baseline. We will cover the evaluation platform, the different benchmarks on which we will be testing the methods and the custom benchmarking tool created to effectively test these methods.

3.1 Evaluation Platform

Benchmarking microservices requires a cluster to deploy the microservices. After careful consideration of popular platforms used for microservice deployment, Openshift was selected as a cluster provider. Openshift (Ope, 2022) by Redhat is a hybrid cloud platform service built around Linux containers and orchestrated by Kubernetes. Kubernetes is an open-source container orchestration platform that helps in microservice deployment, scaling and load balancing. It automates key parts of the microservice deployment pipeline due to which it is a popular option in many enterprises to deploy their applications.

Microservices are deployed on the cluster as pods. Each pod consists of a container which runs the microservice image. A pod can have multiple containers. Pods are by default given some virtual CPUs and memory which can be increased or limited depending on the configuration. Communication from one pod to the other is enabled with the help of the Service configuration. The Service configuration defines the ports that are open and any rules that surround the same.

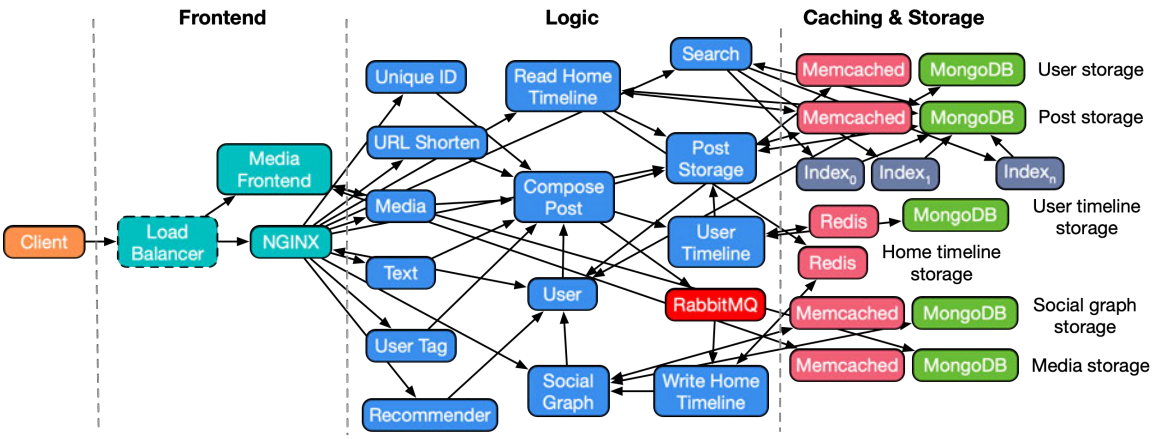


Figure 3·1: Social Network Benchmark Architecture (Dea, 2022)

3.2 Benchmarks

Benchmarks are used to evaluate a system and access relative performance. In the case of microservices, we have multiple benchmarks that can be run. In order to understand the effects of tracing and service mesh on a fixed set of microservices, we run a few selected benchmarks. These include two benchmarks from the DeathStarBench (Dea, 2022) project, Social Network and Hotel Reservation, and the Online Boutique benchmark from Google (Onl, 2022).

3.2.1 Social Network

This benchmark emulates a social network service that consists of multiple loosely coupled microservices that communicate with each other through `Nginx` (ngi, 2022) and `Apache Thrift` (apa, 2022). It consists of 25 individual pods, each representing a part of the social network like users, timelines, posts and media. The benchmark comes with the wrk2 client (wrk, 2022), a tool used to apply load on the microservices and record its latency. API endpoints for functions such as creating and reading posts, reading user information and many more can be utilized. Figure 3·1 shows all the microservices involved in the benchmark.

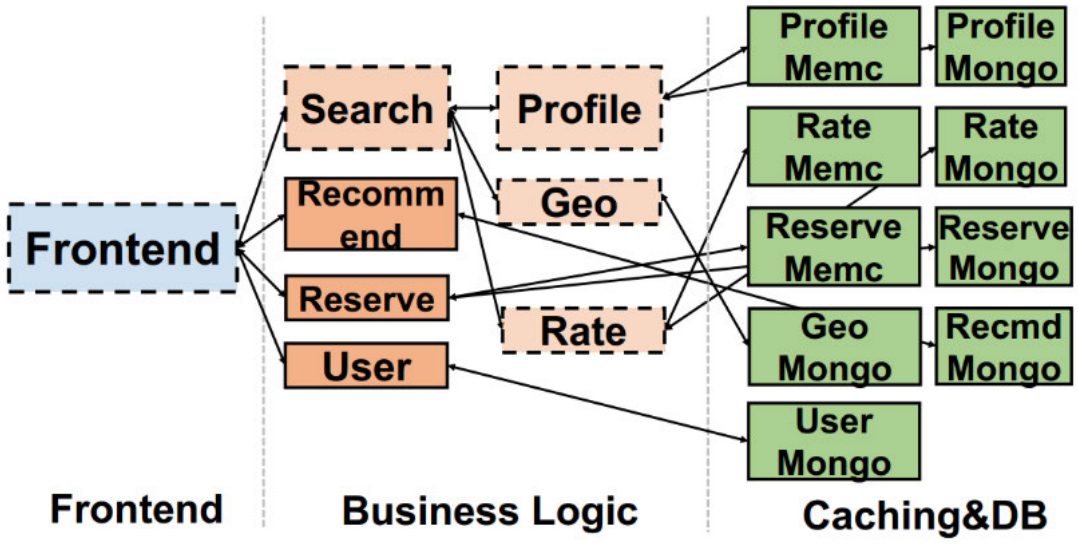


Figure 3·2: Hotel Reservation Benchmark Architecture (Dea, 2022)

3.2.2 Hotel Reservation

This benchmark emulates a hotel reservation service. It is composed of 20 pods that are built on Go and gRPC. As the social network benchmark, it comes with a wrk2 client and a few scripts for applying load on the services and measuring its latency. Figure 3·2 shows all the microservices involved in the benchmark.

3.2.3 Online Boutique

This benchmark is currently maintained by Google. It is a cloud-first online shopping application which consists of 11 microservices that utilize multiple languages. It is built with easily removable components, so testing with multiple tracing applications and service meshes does not require extensive setup. Figure 3·3 shows all the microservices involved in the benchmark.

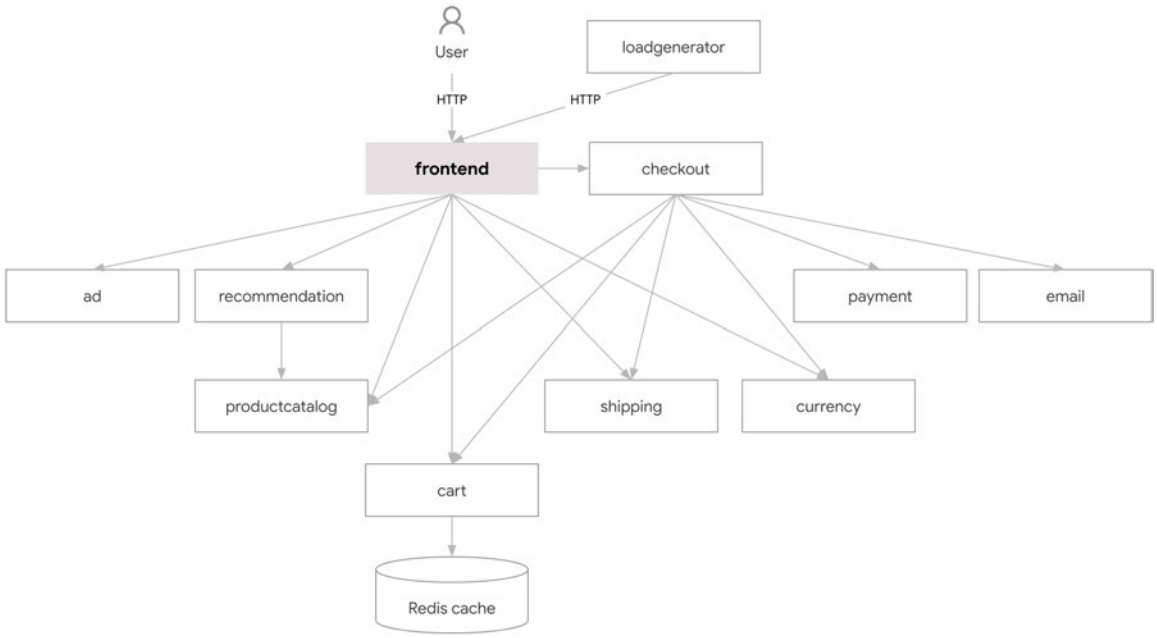


Figure 3·3: Online Boutique Benchmark Architecture (Onl, 2022)

3.3 Unified Benchmark Runner (UBR)

In order to have uniform benchmarks, a tool called Unified Benchmark Runner was developed that would generalize the benchmark process by taking the necessary parameters. The tool is written in such a way that new benchmarks can be added with minor modifications. The tool needs 5 parameters:

- **Threads**: The number of threads used by the client to issue requests.
- **Connections**: The number of connections or users to be simulated. They will be evenly spread amongst the threads specified and will issue requests.
- **Duration**: The amount of time the benchmark will be run for.
- **Requests per Second**: The cap requests per second the client should sustain for the duration of time.
- **Start from scratch**: Yes or no flag for redeploying the benchmark if required.

- **Benchmark selection**: The benchmark to be used. Choices are social-network, hotel-reservation and online-boutique
- **Type of test**: Choice of the type of test to run on the benchmark. This includes running with no tracing, with TCP Dump, with Jaeger or with Istio service mesh. Figure 3·4 shows the different tests.

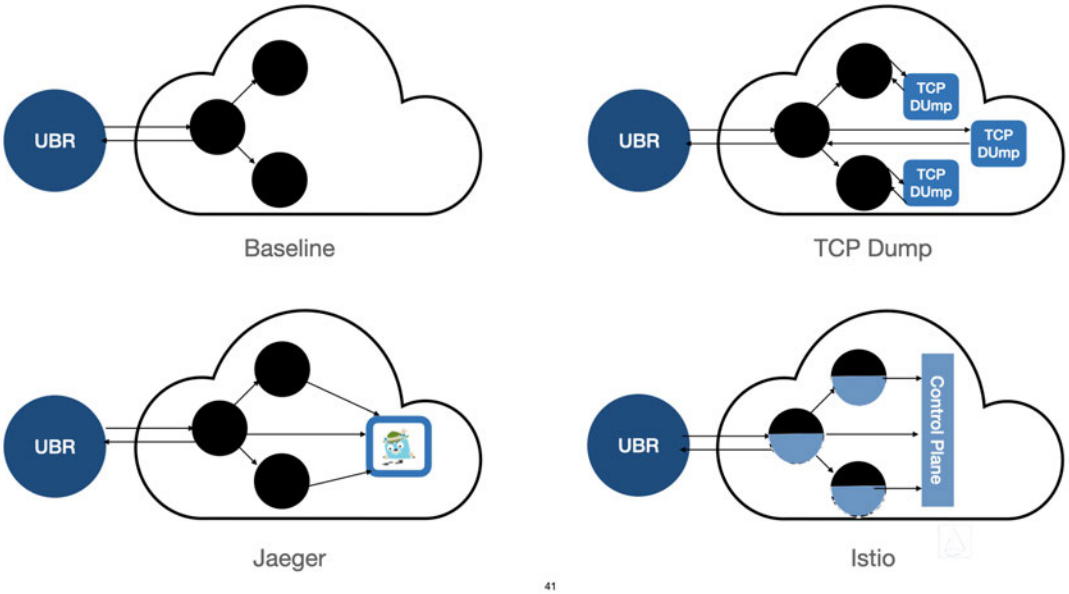
We can run the benchmark as shown below. If we want to run the social-network for 2 threads and 2 connections/users for 60 seconds sustaining 2000 requests per second :

```
./run-benchmark.sh 2 2 60 2000 y social-network
```

The benchmark produces a latency chart which can be then visualized as a CDF plot. Using the benchmark script we can run multiple experiments with relative ease and collect results for analysis purposes. The script ensures that the environments are cleaned up post-experimentation and makes sure that all required files are cloned and ready for the script.

Through multiple experiments, we want to compare 3 main aspects:

- **Latency overhead**: How much of a latency overhead does each of the tracing and service mesh implementations have on the baseline?
- **Throughput overhead**: How much of a throughput overhead does each of the tracing and service mesh implementations have on the baseline?
- **Resource overhead**: How much additional resources are needed to achieve the same performance as the baseline?



Legend

	Pod
TCP DUmp	TCP Dump Pod
	Jaeger Pod
	Pod with Istio Sidecar

Figure 3·4: Different tests run on the benchmark

Chapter 4

Experiments

The experiments were run on an Openshift cluster that was provisioned on Amazon EC2 instances. The cluster had 3 control plane nodes, 2 infrastructure nodes and 3 compute nodes. The compute nodes are where the pods will be provisioned so in order to get enough CPU and memory, the m5.8xlarge flavour of EC2 instances with 32 CPUs and 128 GB memory each were chosen.

As part of the benchmark setup, we made sure that there are no limits being placed on the CPU and memory of the individual microservices. This will allow Openshift to provide as many resources as the microservice requires. Only Istio sidecars have limits to CPU and memory since the Istio setup does not allow disabling these limits. In this case, we set high limits on CPU and memory so that the sidecars are not limited by resources. The Openshift Horizontal Pod Autoscaler was also disabled so as to ensure every microservice runs with a single instance.

4.1 Client Capacity Experiments

First, we need to understand the capacity of the wrk2 client. As mentioned, the wrk2 client is responsible for creating requests to the microservices in the benchmark. In order to understand the maximum supported throughput by wrk2 a few experiments were run. The experiment has 2 pods:

- one running the wrk2 client with that is assigned a number of cores as per the experiment

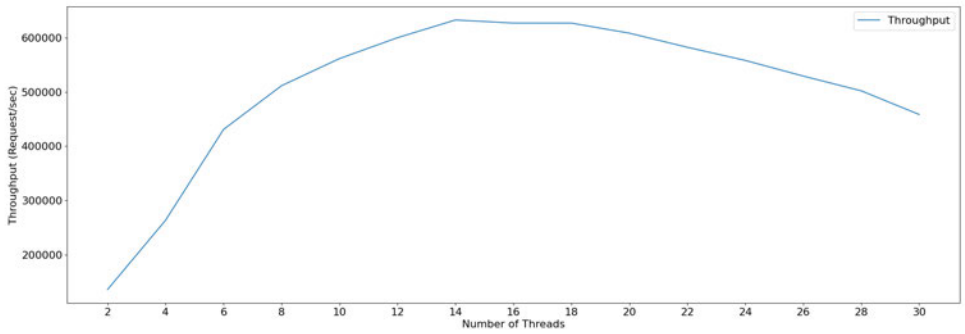


Figure 4·1: Number of threads to the maximum throughput achieved

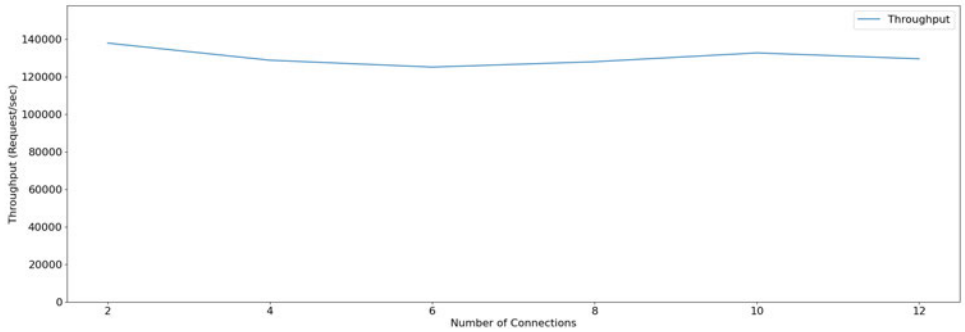


Figure 4·2: Number of connections to the maximum throughput achieved for 2 threads

- one running a hello-world Go server with 30 cores and 20 GB of memory. A request to the pod returns a "hello world" string.

A shell script runs running wrk2 with threads ranging from 2 to 30. An equal number of cores is assigned to the client to ensure every thread gets its own core.

Figure 4·1 shows the graph of the number of threads to the maximum achieved throughput by wrk2. As shown the maximum throughput peaks at 632000 requests per second at 14 threads post which starts declining at a very slow rate. We can infer from this experiment that wrk2 is able to generate high throughput of requests provided the client is able to process them. The throughput is limited by the testing application's capacity after which the throughput starts declining.

The next experiment that was run is fixing the number of threads to 2 and increasing the number of connections. This would help us understand what is the effect of connections on the number of requests that can be generated by the thread. We observed that since the hello-world application is extremely fast, we reached the maximum throughput the client can sustain with a single thread. Figure 4·2 shows that the throughput remains almost constant as the number of connections increases for a fixed thread count of 2. We understand from this that the throughput depends on the number of threads for low-latency applications. The number of connections, which represents the number of users, will have more effect if the latency of the application is more due to which one connection would not be able to generate the maximum throughput of the thread.

4.2 Throughput Experiments

The next set of experiments is to understand the throughput overhead that each of the tracing and service mesh methods have on the overall application. For this, we run the following 4 methods: **no tracing (baseline), TCP Dump, Jaeger and Istio**, on the 3 known benchmarks: **Social Network, Hotel Reservation and Online Boutique**. The results of the experiments on each of the benchmarks are detailed below. The experiments are run using the wrk2 client by fixing the number of threads to 2 and increasing the number of connections up to the arbitrary value of 40960 connections. The maximum throughput achieved in each case is recorded and utilized for further analysis.

4.2.1 Social Network

Figure 4·3 shows the maximum throughput achieved for the thread count of 2 and a fixed number of connections. We can make the following observations from the figure:

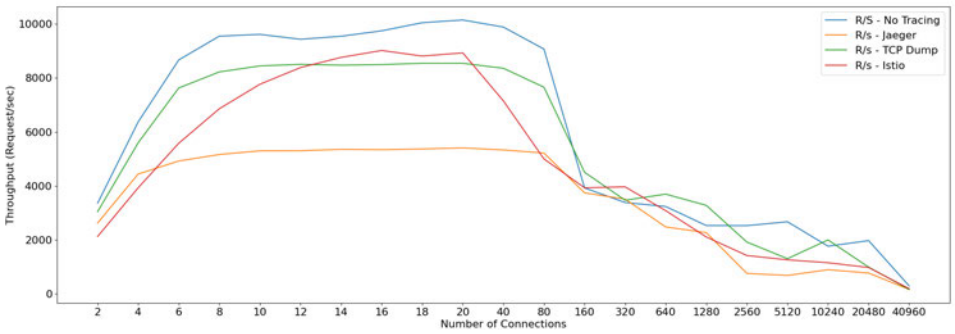


Figure 4·3: Number of connections to the maximum throughput achieved for 2 threads for each method on the social network benchmark

- No tracing (baseline) has the highest throughput. This is followed by Istio and TCP dump with similar second-highest throughput. Jaeger has the lowest out of the four methods.
- We also see that Istio has a low throughput for a lower number of connections and gradually reaches the peak at 18 connections. The rest of the methods achieve a stable throughput of around 8 connections and doesn't show much change post that.
- All the methods start showing a decline in throughput between 20 to 40 connections.

From these experiments, we see that Jaeger has close to a $\sim$**50%** overhead in terms of throughput compared to the baseline while TCP Dump and Istio have close to $\sim$**15%** overhead. The key points we understood from this experiment are as follows:

- *TCP Dump has very low overhead*: This is because TCP Dump records TCP packets which are much smaller in size. Also, every pod has its dedicated TCP Dump pod due to which there is no contention in terms of memory as well. The analysis of the data in the TCP Dump pods is done later by an external service due to which overhead of analysis is also not present.

- *Jaeger has a very high overhead*: Investigating the reason for this we found 2 issues: the first is that due to the increased number of microservices in the social-network benchmark, Jaeger produces a higher amount of data which is handled by a single Jaeger pod. This leads to network saturation and a slowdown of the entire application. The second is that the benchmark by itself doesn't support multiple instances of Jaeger. In our tests with multiple instances of Jaeger, it is seen that only a few instances of the service are used leading to the same network saturation. We conducted experiments by changing the Jaeger sampling rate from 100% to 50%, 25% and 10%. Figure 4·4 shows the throughput obtained under different sampling rates. At 10% sampling, the throughput overhead from Jaeger is reduced to 15%

- *Istio has low overhead*: For Istio the total CPU and memory assigned to the pod are shared amongst the sidecar proxy and the main container. Istio sets a default limit of 2 CPUs for the sidecar. As a service mesh, the requests first come to the sidecar due to which at a higher load, the sidecar becomes a bottleneck. For the social network benchmark, the throughput that can be handled is fairly within the range of the sidecar CPU capacity. As seen in Figures 4·5 and 4·6, the CPU profiles of no tracing and Istio we can see that the CPU usage is almost similar as well

4.2.2 Hotel Reservation

Figure 4·7 shows the maximum throughput achieved for the thread count of 2 and a fixed number of connections for the hotel reservation benchmark. We can make the following observations from the figure:

- Just as the other benchmark the no tracing (baseline) has the highest throughput. This is followed by Jaeger and TCP Dump with similar second-highest

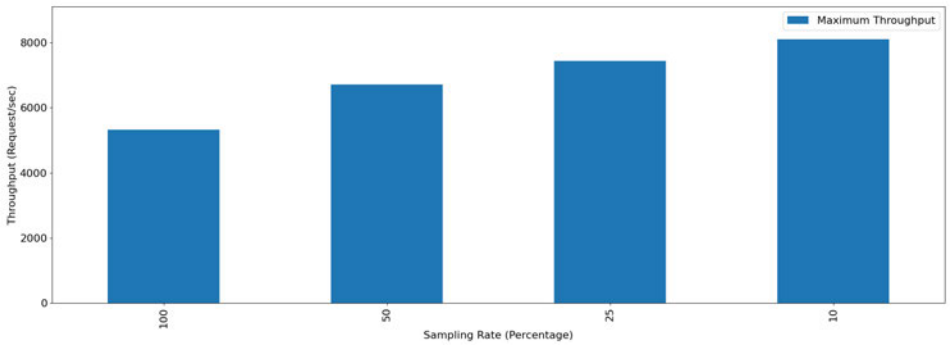


Figure 4·4: The maximum throughput achieved when running the social network benchmark with Jaeger for different sampling rates

throughput. Istio has the lowest out of the 4 methods.

- We see that compared to the social network benchmark Istio has the lowest throughput at close to ∼2000 requests per second
- The maximum throughput is achieved between 160 to 320 connections.

From these experiments, we see that Istio has close to ∼**88%** throughput overhead, Jaeger has ∼**13%** overhead and TCP Dump has a ∼**4%** overhead. The key points we understood from this experiment are as follows:

- *TCP Dump has low overhead*: The reason for this is similar to what we observed in the social network experiment in Section 4.2.1
- *Jaeger has low overhead*: Due to the lower number of microservices as well as lesser span data that is being sent to the Jaeger agent, even at a 100% sampling rate, there is no bottleneck being produced.
- *Istio has high overhead*: For hotel reservations, due to the lower number of services, in baseline with no bottlenecks, the services can scale and use as many resources as possible to generate much higher throughput. But due to the sidecar's CPU limitations, the main container is not able to perform as well.

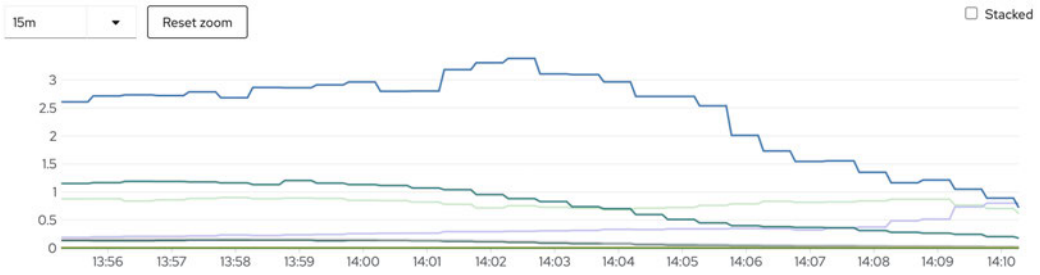


Figure 4·5: No tracing CPU Profile

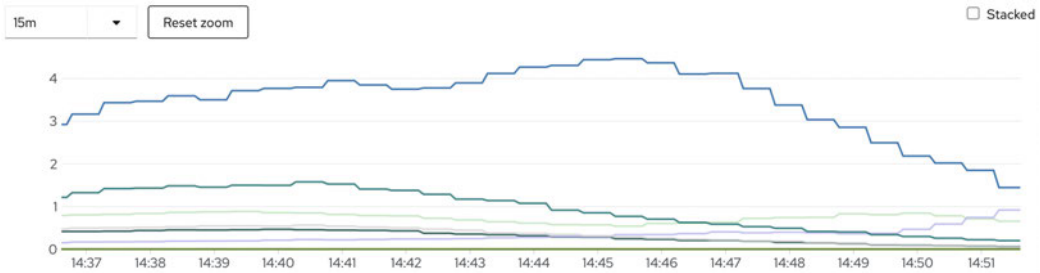


Figure 4·6: Istio CPU Profile

CPU profiles for Social Network's microservices

This is clear with the difference in CPU usage in the case of hotel reservation in Figures 4·8 and 4·9

4.2.3 Online Boutique

Figure 4·10 shows the maximum throughput achieved for the thread count of 2 and a fixed number of connections for the online boutique benchmark. We can make the following inferences from the figure:

- Just as the other benchmark the no tracing (baseline) has the highest throughput. This is followed by Jaeger and TCP Dump with similar second-highest throughput. Istio has the lowest out of the 4 methods.
- We see that compared to the social network benchmark Istio has the lowest throughput at close to ∼350 requests per second

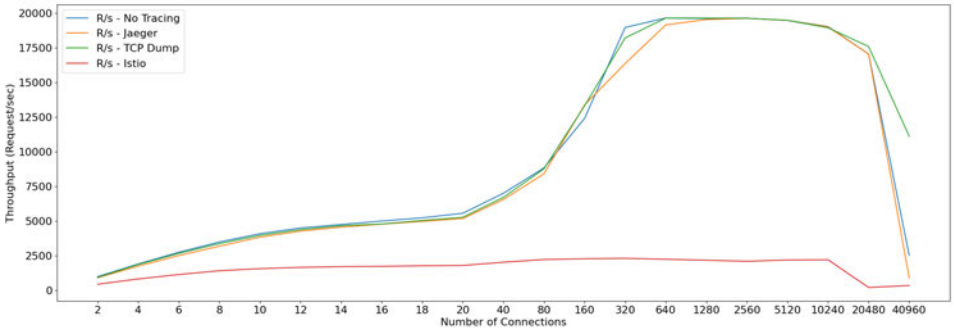


Figure 4·7: Number of connections to the maximum throughput achieved for 2 threads for each method on the hotel reservation benchmark

- The maximum throughput is achieved at 320 connections.
- We also see the maximum throughput is fairly low compared to the other benchmarks coming at 867 requests per second

From these experiments, we see that Istio has close to ∼**60%** throughput overhead, Jaeger has ∼**10%** overhead and TCP Dump has a ∼**9%** overhead. The reason for these results is similar to what is observed in the hotel-reservation throughput experiments in Section 4.2.2 We also noticed that the throughput is fairly low compared to the other two benchmarks. There are two reasons for this: the first is that for online boutique the APIs return the entire web page. This increases the data transfer to almost 7.2 MBps compared to 400 Kbps for the other benchmark, leading to a loss of throughput. The second is that communication in the benchmark is done through GRPC requests. This adds an extra level of overhead compared to simple HTTP requests. A combination of these two factors leads to a decrease in the throughput. Increasing the number of replicas of the services increases the throughput as well.

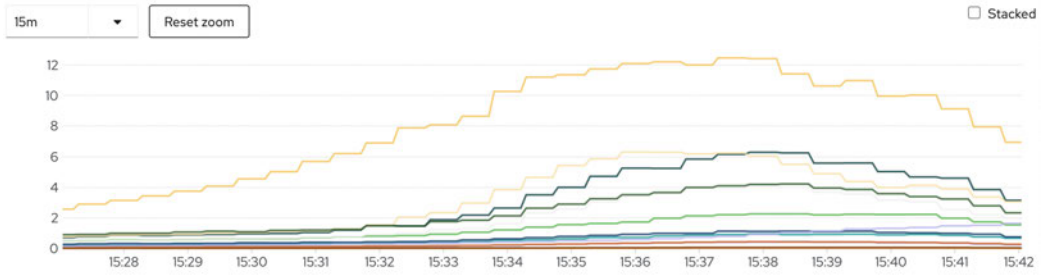


Figure 4·8: No tracing CPU Profile

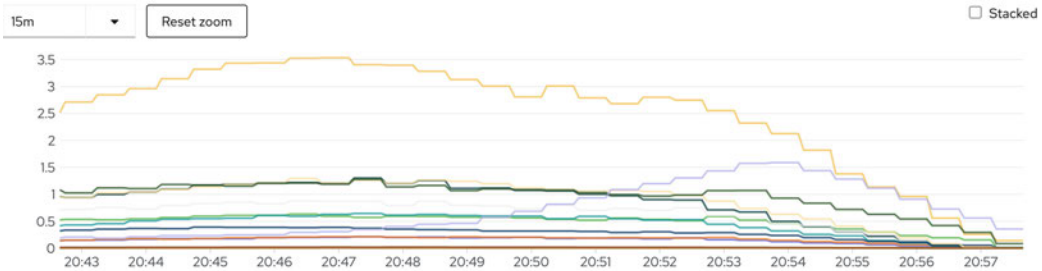


Figure 4·9: Istio CPU Profile

CPU profiles for Hotel Reservation's microservices

4.3 Latency Experiments

For the latency experiments, we ran the benchmarks at a fixed thread, connection and throughput for each of the 4 methods: baseline, Jaeger, TCP Dump and Istio. The data was then represented as a Cumulative Distribution Function (CDF) plot so as to analyze the overhead for each method in terms of latency. A throughput value feasible for all the methods was selected based on the previous experiments. Summary of the experiments can be found in Table 4.2

4.3.1 Social Network

For the social network benchmark, a throughput of 5000 requests per second was selected since it is close to the highest throughput that can be sustained by Jaeger. As shown in Figure 4·11 we can see that Istio and the baseline have almost similar latency, followed by TCP Dump with **∼16%** overhead and Jaeger with having almost **4x**

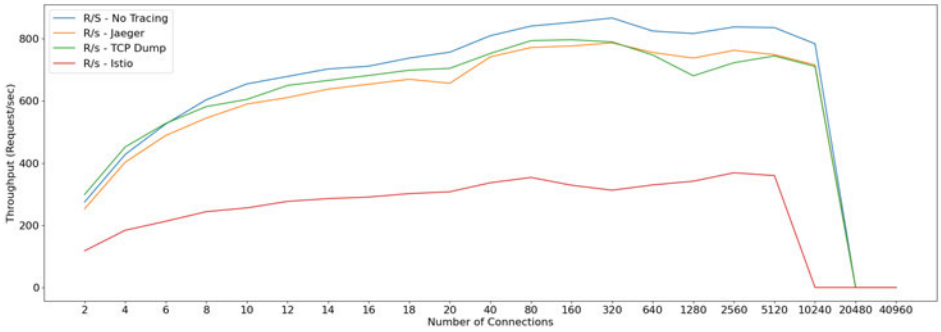


Figure 4·10: Number of connections to the maximum throughput achieved for 2 threads for each method on the online boutique benchmark

Table 4.1: Throughput experiments Summary

	Social Network	**Hotel Reservation**	**Online Boutique**
Number of Services	31	21	11
Maximum Throughput			
Baseline	10k	20k	850
TCP Dump	8.5k	19k	797
Jaeger	5.4k	19k	787
Istio	9k	3.5k	363

latency increase. The reason for this high overhead of Jaeger and the low overhead of Istio is detailed in section 4.2.1. Similar to the throughput experiments, we conducted multiple latency experiments with different Jaeger sampling rates. Figure 4·12 shows the CDF plot while running at sampling rates of 100%, 50%, 25% and 10%. We can see that at 10% sampling the latency overhead comes down to ∼16%

4.3.2 Hotel Reservation

For the Hotel Reservation benchmark, a throughput of 2000 requests per second was selected since it is close to the highest throughput that can be sustained by Istio. As shown in Figure 4·13 we can see that TCP Dump and the baseline have almost similar latency, followed by Jaeger with **∼5%** overhead and Istio with having almost **2x** latency increase. The reason for this high overhead of Istio and the low overhead

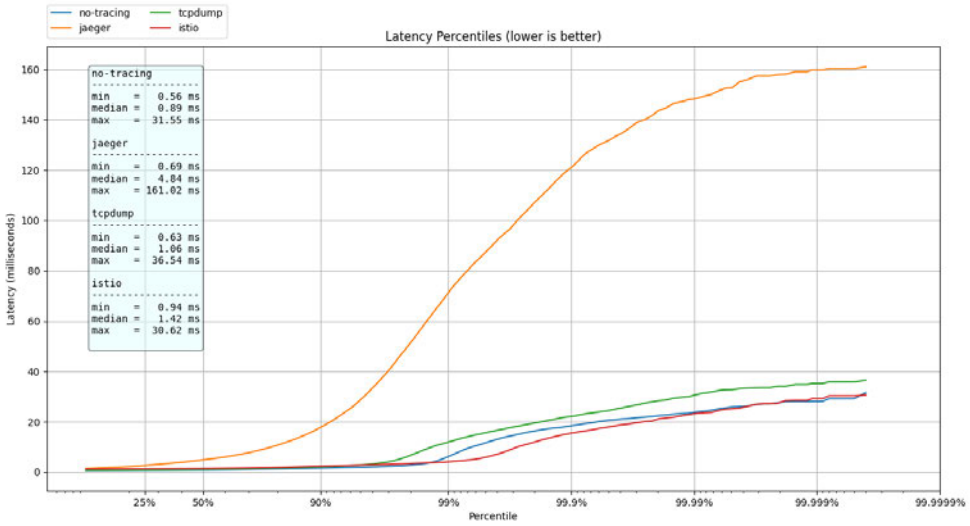


Figure 4·11: CDF plot of latency for all 4 methods on the social network benchmark

of other methods is detailed in section 4.2.2

4.3.3 Online Boutique

For the Online Boutique benchmark, a throughput of 300 requests per second was selected since it is close to the highest throughput that can be sustained by Istio. As shown in Figure 4·14 we can see that TCP Dump and the baseline have almost similar latency, followed by Jaeger with ∼**33%** overhead and Istio with having almost **10x** latency increase. The reason for this high overhead of Istio and the low overhead of other methods is detailed in section 4.2.3

4.4 Resource Allocation Experiments

As seen from the throughput and latency experiments, Jaeger and Istio are constrained by the amount of CPUs, memory or the number of instances (replicas) of the service. We tried to understand these constraints so as to derive some conclusions from them

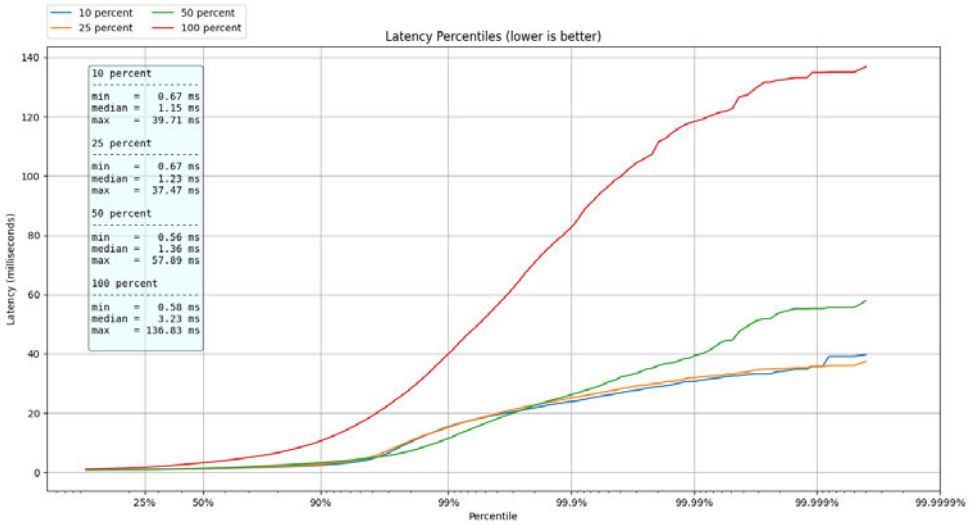


Figure 4·12: The CDF plot obtained when running the social network benchmark with Jaeger for different sampling rates

- *Jaeger resource allocation:* Jaeger runs on a single pod where the data is being accumulated and analyzed. An issue noticed is that as the number of microservices feeding data to Jaeger increases, the amount of data used by the pod is more and the network starts getting saturated. The right method to ensure higher throughput is to increase the number of replicas of Jaeger. Currently increasing the number of replicas of Jaeger in social network doesn't help since it looks to be caching the instance of Jaeger, due to which the same instance is used instead of all the available replicas.

- *Istio resource allocation:* For Istio, there are two resources that can be manipulated: the control plane and the data plane resources. Since the data plane (sidecar) contributes most to the throughput overhead we focused on the resources assigned to the sidecar. We see that increasing the sidecar resources by 2 times there is a 6-7% increase in the throughput. Any increase after that starts degrading performance. This would suggest that a CPU contention is occurring

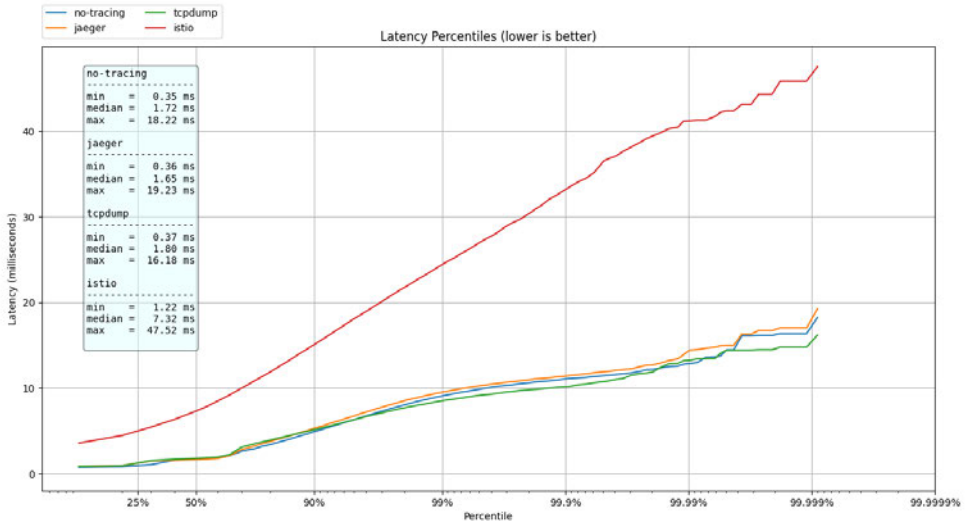


Figure 4·13: CDF plot of latency for all 4 methods on the Hotel Reservation benchmark

as the number of resources is limited. The CPU profiles of hotel reservation benchmark for baseline and running Istio in Figure 4·8 and 4·9 shows that despite giving more resources, Istio doesn't use them due to the contention. We then proceeded to increase the replicas of the services. Increasing the replicas by 1 increases the throughput by 2×.

4.5 Lessons Learnt

During the setup of the benchmarks for experiments, there we a few issues that we faced and fixed so as to get consistent results from the experiments

- *Benchmarking client (wrk2) issues*: When we first started using the benchmarking client wrk2, we found that it was giving inconsistent results. On going through the code we realized that the tool itself gives consistent results at higher requests per second. In order to understand the maximum possible throughput that can be achieved we ran the client capacity tests which helped us narrow

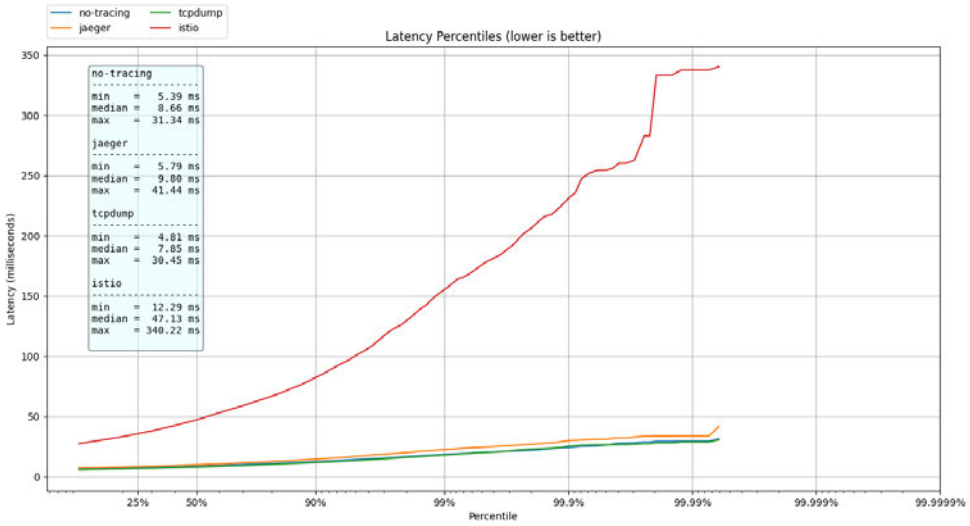


Figure 4·14: CDF plot of latency for all 4 methods on the Online Boutique benchmark

down the possible values of throughput we should be using for the next experiments.

- *Image issues in benchmarks:* The images used by the benchmarks for its microservices had issues due to which we were unable to deploy them. Understanding the commits on the benchmarks we were able to find the last stable version of the benchmarks which we started using for the experiments. Issues were raised on their public repository as well to ensure they would be updated by the benchmark developers in future iterations.

- *Toggling Jaeger:* Enabling and disabling tracing methods was crucial for the experiments. For most benchmarks, Jaeger was built-in into the system and disabling it was an issue we faced. Understanding the code of the benchmark and adding new configuration files to it helped enable and disable Jaeger based on the user input.

- *Resource contention:* The benchmarks require a high number of resources to

Table 4.2: Latency experiments Summary

	Social Network	**Hotel Reservation**	**Online Boutique**
Number of Services	31	21	11
Selected Throughput	5000	2000	300
99th Percentile Latency (ms)			
Baseline	4.01	9.08	18.35
TCP Dump	4.72	9.1	18.879
Jaeger	71.4	9.1	22.21
Istio	4.1	24.4	155

perform without any content of resources. Testing multiple setups we came to the conclusion that every microservice would need at least 2 CPUs and 2GB of RAM so as to not have any contention and give accurate results of overheads.

- *Istio challenges:* Deploying Istio in the default mode will enable tracing and telemetry by default. This would be an extra overhead and would interfere with the results of the experiments. After going through Istio documentation, we were able to tweak the default configuration of Istio so that we can control the features that get deployed along with Istio and also control the sidecar memory and CPU configuration. We also understood about the different annotations that are present in Istio so that we can override the settings of a single microservice in the benchmark.

Chapter 5

Conclusions

The current experiments aimed at understanding the overheads produced by different tracing methods and service mesh architectures on microservices. Over the course of the experiments, we confirmed that often times these tools add considerable overhead to the baseline. In terms of throughput, they add 4-88% overhead and in terms of latency, they add 5-400 % increase. Depending on the use case, these overheads need to be kept in mind so as to ensure that they are accounted for when calculating the Service level availability (SLA) of the application. Below we summarize the results of our experimental evaluation:

- TCP Dump is fairly lightweight, adding throughput overhead between 4%-15% and latency overhead between 5%-33% to the microservices, depending on the benchmark, since it only does the collection of packets.

- Jaeger's overhead heavily depends on the number of microservices and the amount of data sent to the agent. If the application has more microservices, a sufficient number of Jaeger agents must be deployed to ensure the overhead is within an acceptable range. We can also control the sampling rate of Jaeger to reduce the amount of data that is sent to the agent.

- Istio has throughput overhead between 15%-88% and latency overhead between 16%-400% depending on the benchmark it is run on. Based on our initial investigations, it looks like the sidecars become the bottleneck, despite providing

the maximum amount of CPU and memory. Further investigation is necessary to understand why Istio introduces such overhead and whether better tuning can help improve performance.

5.1 Future Work

- *Using TCP Dump for tracing:* TCP Dump is a low overhead tool. If we are able to inject causal information into the packets themselves, we can analyse the collected packets from TCP Dump to have a fairly low overhead system for distributed tracing.

- *Understanding Istio performance:* From our experiments we were able to understand that Istio adds overhead, mainly due to its sidecar architecture and the resources it requires. But we are yet to understand what causes the network saturation and high memory consumption on Istio's side under default settings. A thorough performance analysis of Istio can help understand how it runs under the hood.

- *Optimizing Istio performance:* Using the information obtained from understanding Istio's performance, we can further work on improving its architecture. One of the improvements Istio is working on is a sidecar-less version of the service mesh (dubbed ambient mesh) which would help with better throughput and security.

References

(2022). Apache thrift. https://thrift.apache.org/.

(2022). Deathstarbench. https://github.com/delimitrou/DeathStarBench.

(2022). Istio. https://istio.io/.

(2022). Jaeger. https://www.jaegertracing.io/.

(2022). Ksniff. https://github.com/eldadru/ksniff.

(2022). Linkerd. https://linkerd.io/.

(2022). Nginx. https://www.nginx.com/.

(2022). Online boutique. https://github.com/GoogleCloudPlatform/microservices-demo.

(2022). Red hat openshift. https://www.redhat.com/en/technologies/cloud-computing/openshift.

(2022). Tcpdump. https://www.tcpdump.org/manpages/tcpdump.1.html.

(2022). Wrk2. https://github.com/giltene/wrk2.

Baarzi, A. F. and Kesidis, G. (2021). Showar: Right-sizing and efficient scheduling of microservices. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 427–441.

Sambasivan, R. R., Fonseca, R., Shafer, I., and Ganger, G. R. (2014). So, you want to trace your distributed system? key design insights from years of practical experience. *Parallel Data Lab., Carnegie Mellon Univ., Pittsburgh, PA, USA, Tech. Rep. CMU-PDL*, 14.

Zhang, J., Ferydouni, R., Montana, A., Bittman, D., and Alvaro, P. (2021). 3milebeach: A tracer with teeth. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 458–472.

Zhang, Z., Ramanathan, M. K., Raj, P., Parwal, A., Sherwood, T., and Chabbi, M. (2022). {CRISP}: Critical path analysis of {Large-Scale} microservice architectures. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 655–672.

Zhao, X., Rodrigues, K., Luo, Y., Yuan, D., and Stumm, M. (2016). {Non-Intrusive} performance profiling for entire software stacks based on the flow reconstruction principle. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 603–618.

Zhao, X., Zhang, Y., Lion, D., Ullah, M. F., Luo, Y., Yuan, D., and Stumm, M. (2014). lprof: A non-intrusive request flow profiler for distributed systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 629–644.

Zhu, X., She, G., Xue, B., Zhang, Y., Zhang, Y., Zou, X. K., Duan, X., He, P., Krishnamurthy, A., Lentz, M., et al. (2022). Dissecting service mesh overheads. *arXiv preprint arXiv:2207.00592*.

CURRICULUM VITAE