# Instrumenting Observability in a Decentralized Microservice Architecture

by

Helen X. Liu

S.B. Computer Science and Engineering and Business Analytics, MIT, 2023

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2024



| | |
|---|---|
| Authored by: | Helen X. Liu<br>Department of Electrical Engineering and Computer Science<br>August 9, 2024 |
| Certified by: | Michael Cafarella<br>Principal Research Scientist at MIT CSAIL, Thesis Supervisor |
| Certified by: | Aleks Ryabin<br>Director of Engineering at NetApp, Inc., Thesis Supervisor |
| Accepted by: | Katrina LaCurts<br>Chair<br>Master of Engineering Thesis Committee |

# Instrumenting Observability in a Decentralized Microservice Architecture

by

Helen X. Liu

Submitted to the Department of Electrical Engineering and Computer Science
on August 9, 2024 in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

## ABSTRACT

Software systems have increased in complexity over time, and with this increased complexity has come an increased need to keep these systems organized and functioning efficiently. Observability is closely attached to ensuring this correct and effective system function. Without system monitoring, it is difficult to pinpoint when errors occur and correct them at their sources. Monitoring systems also helps to understand a system from the outside by allowing developers to ask questions about the system's state and function without needing to know the details of what comprises the system's internal behavior. While there are existing solutions for observability frameworks, these solutions do not target microservice architectures, which are used more and more with expansive code bases, such as those likely to be employed in an industry environment. They also require extensive configuration to be fully integrated with a pre-existing system. As such, the challenge lies primarily in adapting observability solutions to a decentralized, microservice architecture found in an industry setting. The existing solutions also come with advantages and disadvantages for different situations, so they are often incomplete in addressing an entire system's needs. The integrated system created here satisfies our system's requirements of a consolidated observability platform while also enabling future customizations, thereby allowing problems to be identified more quickly and proactively.

Thesis supervisor: Michael Cafarella
Title: Principal Research Scientist at MIT CSAIL

Thesis supervisor: Aleks Ryabin
Title: Director of Engineering at NetApp, Inc.

# Acknowledgments

I would like to thank NetApp for all the support and guidance I received on this thesis project during my time there. More specifically, I would like to give a special thanks to Narasimha Reddy, for unfailingly taking time out of his day to assist me on any and all problems I ran into; to Ajay Aggarwal, for consistently providing me with feedback and a sounding board for my ideas; to Aleks Ryabin, for always asking the critical questions I needed to answer to guide my project in the right direction; along with everyone else that I met who eagerly helped me in any way they were able to at any time. I felt welcomed with open arms the entire time I was hosted at NetApp, and the warmth of all the people there made my time so pleasant. Without everybody's insights, I would not have completed anywhere as much as I did in my project.

I would also like to thank Michael Cafarella for assisting on and advising this thesis. I would like to thank the 6-A program and the EECS department as a whole for providing me with the resources to extend my time at MIT and complete my graduate education here.

Finally, I would like to thank my family for all of their constant love and support, without which I would not be where I am today. Thank you to my parents and my sisters for always being my biggest supporters.

# Biographical Sketch

Helen Liu received her S.B. degree in Computer Science and Engineering and Business Analytics from MIT in 2023. She then continued her studies at MIT as a Master of Engineering student in Computer Science, graduating in September 2024 upon acceptance of this thesis.

She has previously participated in numerous research projects. Her first exposure to research being in materials science, for which she is a second author for a paper published in the *Nature Materials* scholastic magazine, "Scalable optical manufacture of dynamic structural color in stretchable materials" [1]. Since then, she has completed research in updating models to screen for pulmonary diseases, designing a web application for reducing inconsistencies and polarization in the media, and predicting specific images of gene expression given other, less expensive images of the same genes.

Her research experience culminates in this research project centered around observability in decentralized software systems.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

As software systems become increasingly complex, maintaining their organization and efficiency becomes more critical. Observability plays a vital role in ensuring the proper functioning of these systems, allowing for errors to be detected and corrected. It encompasses three key elements: metrics, logs, and traces, each serving distinct purposes in monitoring system behavior. While frameworks for observability already exist, they often do not cater to the decentralized nature of microservice architectures commonly found in industry settings. Adapting these frameworks to suit such architectures presents a significant challenge. Moreover, existing solutions may be insufficient, as they come with their own sets of advantages and disadvantages, leaving certain system needs unaddressed. Our integrated system addresses these challenges by meeting observability requirements while ensuring flexibility and customizability, facilitating quicker and more proactive problem identification.

## 1.1 Context

A microservices architecture is commonly employed in software systems to speed up application development by allowing developers to work on different sections of a system in parallel while minimizing the risk of interference from each other. It also provides a more clear-cut separation of different functions, creating a more organized architecture as well. A service

[2] is a building block of a microservices architecture system that is designed to carry out one specific function. It handles all tasks related to that function, and it communicates with other services within the architecture to handle more involved problems, such as end-to-end tasks that require many related but distinct pieces.

Let us walk through an example of a user request being executed in a system that employs a microservice architecture. In this fabricated situation, laid out below in Figure 1.1, the user wants to start a subscription to use the web application's product, so they make a request to create this subscription. This request first passes through the service responsible for the web application's user interface (UI). The web UI service then makes API calls to both the service responsible for handling user accounts and the service responsible for handling billing and payments. Each of these services in turn accesses its related database, with the user's account being updated with the new subscription in the accounts database and the new payment for the subscription being logged in the payments database. This example is a simple version of a microservice architecture; in a real-world example, there would be many more services interacting with each other and with other components beyond databases.

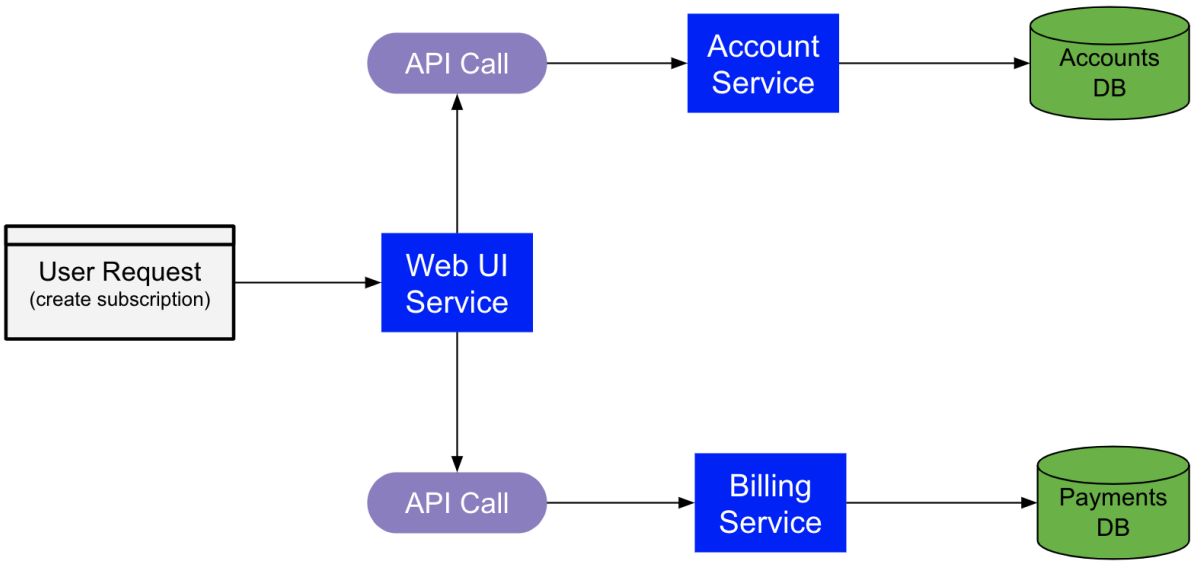


Figure 1.1: Example of a user request executed in a system with a microservice architecture

## 1.2 Problem Statement

The systems we are monitoring in our paper have a few metrics in place already, but these metrics have many limitations. First, they lack granularity and do not exist on a service-by-service basis. Given that a microservice architecture involves extensive communication between services, it is a huge pitfall to only be able to obtain metrics for certain services and not others, and it is even more of a problem to have all these metrics in different forms and accessed in different places as well, instead of all together in a single, standardized, consolidated interface.

These existing metrics in our systems also do not include any application performance metrics, which monitor the state of different applications. Application performance metrics include metrics on the web server and metrics on memory usage. In an industry setting, especially one in which software is being used by users external to the company, this can be detrimental in assessing the source of user-facing issues that arise and thus detrimental to the business value of the company. In these situations, it is always best to be proactive where possible in identifying and resolving bugs or system failures, ideally before users even realize they are present.

Logs, on the other hand, are already fully present in our systems, but they use a third-party software that does not handle metrics or traces and cannot support many other back-ends. We want to create a cohesive observability framework that can fully incorporate all three components of observability and is flexible in export, so we need to migrate logs to this framework. Since there are no traces currently in our systems, we also need to incorporate traces into this framework from scratch.

Without relevant and specific metrics, it becomes much more difficult to effectively monitor large software systems. The same can be extended to logs and traces. We want all of these observability data points to be service-specific to be able to use them best, and having them all consolidated in one place would allow developers to use them most efficiently.

To extend our example of a user request from the previous section, say we have a developer whose job it is to ensure this web application operates consistently and reliably. They receive a complaint from the user that the user's requests to the application are very slow, so they first want to determine whether the issue is user-specific or system-wide. However, since they do not have any metrics on the performance of the application, they must rely on other users complaining in order to see if the issue is system-wide.

Let us assume there are no other complaints, so the issue is likely user-specific. From here, we want to see if the issue applies to just one or all of the services with which the user is interacting. The developer would be able to figure this out quickly if they had access to service-specific metrics on HTTP request response times, but since they do not, they cannot easily figure out which services are affected. The developer also cannot easily figure out where the problem originates because they do not have access to traces that can show them which components the user request touched and the time spent to execute each call to a component, which would have drastically narrowed down the potential sources of error. Finally, assuming the developer eventually approximately locates where the issue is, they still cannot easily identify which lines of code are the problem because they have to parse through hundreds of lines of log messages to see which log events (emitted from running code) are erroring out and causing requests to take longer. As such, they will take much longer to fix the issue than they would have if they had access to a complete observability platform.

As we have now seen, the current state of the system we are examining is not good enough. The system does not adequately provide sufficient tools for monitoring its different parts, much less gauging its health as a whole. The next section will elaborate on existing solutions for observability frameworks that accomplish the same goals as the ones we aim to reach, but these solutions do not target microservice architectures, which are increasingly used in industry settings. As such, the goal of this research is to adapt existing observability solutions to a decentralized, microservice architecture. In this architecture, scalability is very

important as application development speeds up and services are created to fulfill new needs that arise. If developers needed to manually configure metrics in every new service that was created, it would be very costly from a time (and thus, money) perspective. Flexibility is also crucial, for each service does not have identical needs to every other service, thus requiring a highly customizable framework that can be tailored to a specific service whenever necessary. With these goals in mind, we aim to create an observability framework that can thus identify problems more quickly and proactively.

## 1.3 Existing Solutions

As mentioned previously, there are numerous existing solutions for observability already. In this section, we will introduce the relevant solutions to our paper.

Spring Boot [3] is a framework to help build web-based applications. In our system, most services already use Spring Boot. Within the Spring Boot framework is a dependency called Spring Boot Actuator [4], which monitors and manages the web application. This dependency comes pre-loaded with Micrometer [5], facilitating an easy integration of Micrometer with existing Spring Boot services. Micrometer is used for application observability, which is exactly what we want to configure in our systems; it is one of the only such frameworks, making it the most well-documented as well. Micrometer allows users to instrument their code once then export all the observability data to one or more of their chosen backend systems, which aggregate the data they receive into a visual representation, such as a chart or a graph. Micrometer provides much of the functionality we want in our framework, as it provides many default application performance metrics, along with the ability to instrument further custom metrics. For this reason, a couple services in our system already use Micrometer individually.

The main issue with Micrometer is that much of the existing work done with it primarily uses the assumption of a single-service architecture. This work is not conducive to applying

Micrometer to large, pre-existing codebases employing a decentralized microservice architecture (i.e. those often found in industry settings), as the challenge now lies in integrating this framework into what is already there. For this, the OpenTelemetry Collector [6] and OpenTelemetry Protocol [7] are helpful.

OpenTelemetry, like Micrometer, is also a framework for observability that can be exported to any backend system. The relevant aspects of OpenTelemetry to this project are the OpenTelemetry Collector and the OpenTelemetry Protocol (OTLP). The OpenTelemetry Collector acts as a centralized middleman to which data can be easily exported, and it then processes and exports that data in turn to one or more backends. It provides a high degree of flexibility in data export, as it is simple to change the configuration of backends to which data is sent. Meanwhile, the OpenTelemetry Protocol provides a standard shape and format with which data can be exported. Being able to convert data into this format helps with integrating the framework into the existing architectures, which is why we chose to use it with the various components of observability in our framework.

## 1.4 Related Work

With system monitoring and observability being a rapidly developing area, there are a few existing papers that provide a comprehensive overview on frameworks that are currently being used, along with what the most important criteria are to consider when designing a new framework.

Observability is an ever-evolving issue because the systems that need to be observed are constantly changing. One major change was the shift to decentralized microservice architectures from a single-service, monolith architecture. Because of this shift, systems now have a lot more moving parts, i.e. more parts that need to be observed. Another change was the shift to cloud-native technologies. An infrastructure hosted in the cloud results in more flexibility and reliability while enhancing performance simultaneously. However, this shift

affected observability work by requiring it to be more dynamic in order to reflect the more dynamic nature of cloud computing. With these increasingly complex systems, it became more important to have increased visibility into them as well. The various components of observability provide this necessary context and insight into the state and performance of a system, thus minimizing the time it takes for developers to understand what is happening in a system.

In line with these shifted observability needs, Karumuri, Solleza, Zdonik, & Tatbul [8] provide a detailed discussion of how important observability is, especially in an industry setting like the one we have in this paper. They use an incident from Slack as an example of where observability was used to identify the root cause of the issue, and they bring up the very important point of observability also being a data management problem. With all of this observability data, it becomes critical to manage the data well such that site reliability engineers (SREs) can effectively utilize it to monitor systems rather than be overwhelmed and lost in the flood of new data. We have kept this in mind as we developed our framework, and this need for organization is a key part of why we treat metrics, logs, and traces (the three components of observability) separately. However, this was not the primary objective of our research because we found that for our business purposes, it was more important to focus on a different design principle that Karumuri et al. bring up: flexibility in a distributed environment. With the distributed nature of so many software systems used in industry today, our primary goal once we established the framwework was to ensure we could easily apply it to all parts of a system, which we describe some more in the next section.

Niedermaier, Koetter, Freymann, & Wagner [9] interviewed many software professionals to compile a list of key challenges to and requirements for an observability framework. One such challenge was a lack of experience, time, and resources to master and integrate existing technologies. We similarly discovered this challenge at the beginning of our project when we first met with developers to discuss the key features of the observability framework. We thus created the framework that we describe in this paper so that other developers

within the company have an easy way of monitoring the systems without needing extensive experience, time, or resources. By integrating Micrometer and OpenTelemetry only once, and by using these tools that enable data to be exported to any backend, SREs and developers who have little experience with these technologies can access compiled observability data in the monitoring backend, allowing this data to be easily seen and interpreted regardless of experience level. Like Karumuri et al., Niedermaier et al. also expressed concern over the flood of data that comes in from an observability system. Because this is a concern we also share, we performed some calculations on our final framework to estimate how many data points are regularly emitted in chapter 4, where we look at the state of our research on from different perspectives. Finally, a major requirement of observability systems that Niedermaier et al. concluded from their interviews was that it should be an all-in-one solution, which we had also realized ourselves and subsequently prioritized.

In a different paper, Usman, Ferlin, Brunstrom, and Taheri [10] conducted a survey on observability tools for microservices. Their survey reaffirmed the status of the Cloud Native Computing Foundation's (CNCF) open-source OpenTelemetry project as the emerging industry standard for managing observability data. They also mention the SRE's four "golden signals" of monitoring: latency, traffic, errors, and saturation. The SREs we spoke to mentioned these golden signals as well, so we made sure to include them in our evaluation of our framework in chapter 4. Overall, we agreed strongly with Usman et al. in their emphasis of the importance of a unified platform for data, along with easy customizability of the various components involved in the overarching observability framework. They mention decoupling data sources from data sinks, which is what we aim to address in having a separated-out flow of data across various technologies, as seen later in Figure 1.2 in the following section.

The main difficulty in using observability frameworks that people have already created is that each system we want to monitor is different, so what other people have done frequently does not apply to what we want to do now. Every system is unique, and part of the challenge of this research is figuring out how to integrate the open-source frameworks we mentioned

in the previous section with the microservice architecture and custom backend that we are working with in this industry setting.

## 1.5 Design Goals

The existing work with Micrometer has primarily focused on single-service applications. As many companies now employ microservice architectures, it is necessary to expand upon this work to apply the Micrometer framework to these more decentralized systems. Our aim is to ensure that each microservice does not have to implement an observability solution itself, as this is impractical for scalability from the perspective of developers' time. Instead, we want every service to consume a single tool in line with a common framework, with the option to include any desired tags on metrics and override default settings of which metrics are exported. We thus need to configure this framework in a common component across all services.

We also need to ensure future flexibility and customizability. While the Micrometer façade builds in the ability to create custom metrics, extending this ability in a decentralized architecture presents challenges around having a standardized convention of doing so. The idea of flexibility can be applied to exporting observability data. The backend to which data is exported can change over time, and it may also vary from service to service, so we aim to explore what the best strategy is to achieve this flexibility in data export, including potentially using existing solutions such as the OpenTelemetry Collector. One item to consider here is the "push" versus "pull" method of exporting metrics, and the flexibility of both. We will elaborate in this more in subsection 2.2.1.

A further goal for the project is to build in support for dynamic tags. Each meter in Micrometer, keeping track of a specific metric, has a name and a variable number of customizable tags represented as key-value pairs. Micrometer does not currently provide support for tag values to change dynamically as a variable, despite many developers asking

for this feature; tag values are thus represented statically as strings. This is because the Micrometer system maintains an internal map of registered metrics using names and tags, so dynamic tagging would make collisions possible for metrics with the same name to suddenly have the same tags too, and the system would end up merging the two metrics arbitrarily. The static meter tags are thus for the purpose of making the meter registry, where all the meters are stored, collision-resistant.

However, dynamic tags would allow businesses to provide a very helpful level of granularity to the metrics that they desire. For example, many companies have services that are multi-tenant [11], which means that a single software resource can be used by multiple end users or groups. As such, it is important to be able to filter out metrics specific to different tenants using the same resource, but a tenant's identifier is a field that changes between tenants, hence the need for dynamic tagging. We aim to enable dynamic tagging for the most important use cases and circumvent these built-in Micrometer limitations.

Another goal for the project is to extend this observability framework from metrics to OpenTelemetry-based logs and traces. Since observability typically refers to all three components of metrics, logs, and traces, we would like to first lay the foundation for metrics, then subsequently broaden the scope of the framework to logs and traces. For logs and traces, we want to export them in OpenTelemetry format because only after everything is in the same format can we begin to integrate metrics, logs, and traces through contextual correlation, which will help with the challenge of distributed tracing—tracing a related call across services.

When the project is complete, it should look like the flow chart in Figure 1.2. In this figure, we see that there are three flows for the three components of observability that all go into one OpenTelemetry Collector. From there, each component has a different HTTP endpoint on the custom backend product we are using because metrics, logs, and traces data all have different shapes according to the OTLP specifications. However, once they are ingested, they are able to be visualized in the same backend platform. This is only a quick,

high-level overview; we will explain the parts of this flow chart more in-depth in subsequent chapters.

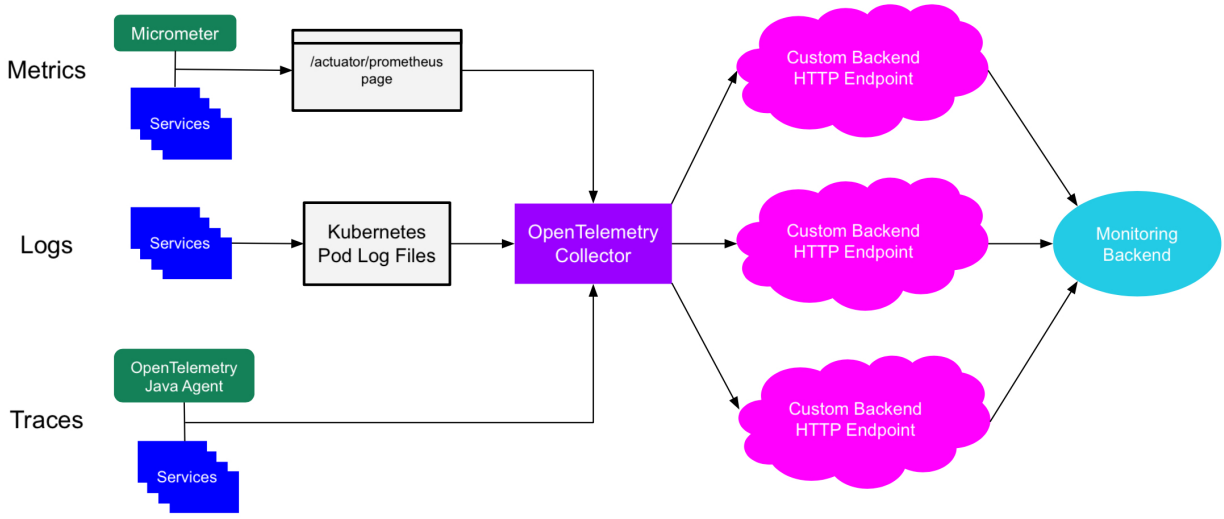


Figure 1.2: Overall final architecture

# Chapter 2

# Metrics

As mentioned previously, the three major components of observability—metrics, logs, and traces—have distinct but vital roles in system monitoring. Metrics refer to specific data combined from measuring events. They offer statistics on various aspects of a system in order to then be used to identify patterns and eventually provide a baseline against which to measure current systems to better sense when something amiss has occurred.

In this chapter, we provide an overview of which metrics are helpful to have, what features of our observability framework regarding metrics are most important, and how we implemented these features in our system.

## 2.1 Metrics and Micrometer Overview

There are many types of metrics, but not all of them are relevant and helpful in an observability context. Furthermore, in the context of a software system that employs a microservice architecture, many of the existing metrics in our system lack sufficient granularity to tunnel into specific services, as they do not exist on a service-to-service basis. They also do not include any application performance metrics, such as metrics on the web server or on memory usage. As such, there is a lack of visibility into these software systems.

When instrumenting metrics into a service, we want to make sure that the method

by which these metrics are instrumented is extensible to other services within the same architecture to maintain a certain degree of standardization. With this in mind, we decided to utilize the Micrometer package, included in Spring Boot Actuator.

Micrometer collects measurements using `Meters` of different types, with the primary meter types being `Timers`, `Gauges`, and `Counters`. As a brief overview, `Timer` objects measure the latency duration and frequency of events, with each `Timer` reporting the total time and count of each event. `Gauge` objects are used as a handle on the current value of a metric, changing only when it samples the measurement it is in charge of and holding no information on what occurs between samples. `Counter` objects provide a count of a metric by only ever incrementing by a fixed, positive amount. A helpful `Meter` object to have in our example from the previous chapter would have been a `Timer` object measuring HTTP request response times. Having this metric would have helped our developer narrow down the user's problem to the specific service(s) and HTTP endpoint(s) that were affected, thereby reducing the scope of the engineer's work.

A few other `Meter` primitives include `DistributionSummary`, `LongTaskTimer`, `FunctionCounter`, `FunctionTimer`, and `TimeGauge`, but these primitives are used less often, so they are not as relevant in our proposed observability framework currently. Regardless of type, each `Meter` is held in a `MeterRegistry`, which holds the most recent value of the meter. Every monitoring system supported by Micrometer has its own `MeterRegistry`. For example, the OpenTelemetry Protocol (OTLP) is supported by Micrometer, so there is an `OtlpMeterRegistry` specific to OTLP.

Micrometer auto-instruments many metrics on various parts of the system by default, depending on which software technologies are being used by the system in which the library is located. For example, our system uses the Apache Log4j2 [12] logger, so there are automatically metrics that measure the log4j2 events emitted, tagged by the different levels for log messages. Other common groupings of metrics include metrics on the Java Virtual Machine, system CPU, HTTP requests, data source connections, threads, disk space, and

the application startup time.

## 2.2 Implementation

Making changes that apply to all services within a microservice architecture is challenging. To remedy this, it is beneficial to have a custom library from which all services inherit. Within this common library, developers can put Spring Boot packages to equip the child services with the Spring Boot framework, helpful for building web-based Spring applications in Java. To add in Micrometer, we added the Spring Boot Actuator package to our common library.

Adding Spring Boot Actuator configures additional functionality beyond having Micrometer available. By including the Actuator package, certain `/actuator` web endpoints become available on the service, with the option to configure more if necessary [4]. The default endpoints are `/health` and `/info`, which show basic application health information and information about the application itself, respectively. Additional endpoints that are helpful for visualizing service-specific metrics are `/metrics` and `/prometheus`, both of which show all the metrics being tracked by Micrometer, with the latter exposing these metrics in a format that can be scraped by a Prometheus server [13]. This `/prometheus` endpoint will be more relevant in the next section, when discussing the manner in which to export each service's metrics to the same backend visualization platform.

The main requirements of our observability framework, decided upon in conjunction with relevant stakeholders looking for expanded observability metrics in the services they are in charge of, are as follows:

- Can be visualized in the existing backend platform
- Can customize how much information is being transmitted
- Provides a standardized way of instrumenting custom metrics specific to a certain service

- Includes tenant ID as a tag on metrics to obtain more granular data on the multitenant services

In the following subsections, we will explain how our implemented solution addresses each of these points.

### 2.2.1 Connection to Backend

As mentioned earlier, the metrics auto-instrumented by Micrometer are readily available at a service's `/actuator/metrics` and `/actuator/prometheus` web endpoints if the endpoints are enabled. From here, the challenge was to figure out how to get the metrics into a visualization platform.

Here we decided to use the OpenTelemetry Collector [6]. A few factors went into this decision:

1. The backend to which data is exported can change over time. We wanted to ensure that we would not need to set up a whole new export mechanism in order to export data to a different backend, should we choose to do so later.

2. We may want to export data to more than one backend. For example, it would be nice for debugging purposes to also export our data to the console to see whether any metrics are being dropped when exporting to our chosen backend product.

3. Where we want to export data may vary from service to service, so we want to provide flexibility and customizability for each service by giving it the option to export to different backends in this microservice architecture.

OpenTelemetry provides the industry standard for observability, as an open source project from the Cloud Native Computing Foundation. Before this, there was no real standard, but OpenTelemetry supplies many different components that altogether fill this hole. The most important components to this project are the OpenTelemetry Protocol (OTLP) [7] and the

OpenTelemetry Collector. OTLP provides a standard shape and format to telemetry data, while the Collector acts as a proxy that can be used to receive, process, and export telemetry data, i.e. metrics, logs, and traces. It consolidates everything in one place, and that makes it very useful when configuring different telemetry data, especially if the telemetry data comes from various sources. OpenTelemetry is designed to be extensible and customizable, which makes it fit well into the scope of this project.

Next, the question arose of whether we wanted a "push" or a "pull" mechanism for exporting our metrics, and we wanted to consider the scalability of both. The "push" method refers to having servers regularly send data to a backend database, which places the responsibility on the servers, while the "pull" method has the backend services regularly querying data from the servers, thereby placing the responsibility on the backend services. In terms of the tradeoffs of each, having a "push" method would require all services to know where to push to, while a "pull" method would offer more flexibility in changing backends but would be more complex to implement, as Micrometer is not configured optimally for this.

Since we decided to use the OpenTelemetry Collector for the above reasons, we actually are able to use a combination of the "push" and the "pull" methods to reap the advantages of both while avoiding their corresponding disadvantages. To "pull", we can have the Collector scrape data from one of the endpoints to which all of a service's metric data is available. Since the Collector has a built-in Prometheus scraping functionality, it is no longer complicated to implement because we only need to make the `/actuator/prometheus` web endpoint available for the Collector to be able to ingest the metrics data. To "push", the Collector then can process this data however we configure it to [14], and the transformed data can be exported to whichever backend system (or systems if multiple) we would like to use. Since the Collector first aggregates all the data from the different services, there is now only one entity pushing to the backend service, which nullifies the problem of having all services know where to push data. Having this Collector be easily configurable helps us satisfy our initial needs

of wanting flexibility in export because we can edit the receivers, processors, and exporters configurations of the Collector to change the backend to which we export data, export data to multiple backends, and customize which services send data to which backends.

Because we are using the OpenTelemetry Collector, the `/actuator/prometheus` endpoint is particularly useful because it exposes all of the metric data on the same web page, as opposed to the `/actuator/metrics` endpoint, which requires indexing into a metric name (i.e. calling a specific `/actuator/metrics/<metric.name>` endpoint) to get a single metric's data. Having all the metric data for a given service in one place is important so that the Collector can pull this data all at once using its built-in Prometheus configuration to act like a Prometheus server scraping data. In this way, the Collector can easily obtain an entire service's metrics efficiently.

Kubernetes [15] is a very commonly used system for microservice architectures, so we utilize it in our project. OpenTelemetry supports Kubernetes very well, as both are open-source projects created by the Cloud Native Computing Foundation. Thus, we will use Kubernetes to deploy the Collector. See Appendix A for an example of a basic Collector configuration using OTLP to receive and export data.

To give a brief overview of how Kubernetes works in our system, Figure 2.1 shows the structure of a single Kubernetes cluster. Within this cluster, represented by the yellow rectangle, are several nodes, represented by pink boxes. Each node provides a copy of the same system, with each node running all of a system's microservices as pods. Each microservice is run on its own pod, and the OpenTelemetry Collector is also run on its own pod.

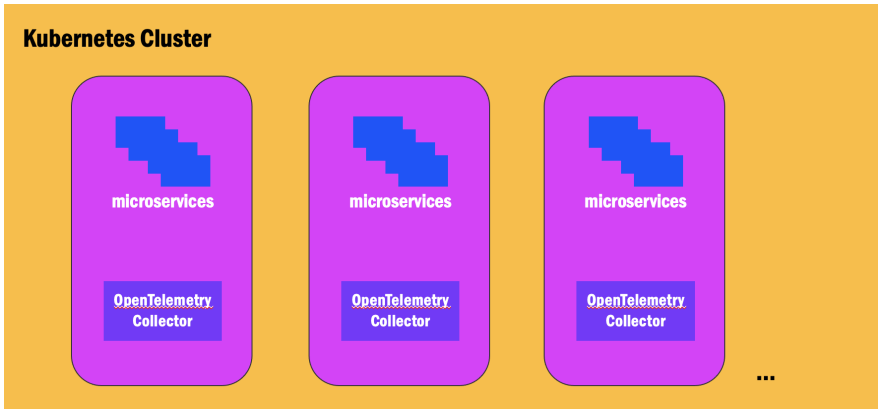


Figure 2.1: A single Kubernetes cluster

Figure 2.1 shows one possible configuration of the OpenTelemetry Collector, although there are multiple possible configurations of the Collector that would result in different diagrams. In this diagram, we have configured the Collector as a Kubernetes daemonset, which means that each node has its own Collector that monitors only the microservices within its node. We decided that this mode was optimal because it ensured that any one Collector would not confuse the information collected from one service with another copy of the service on another node.

As shown in Figure 2.2, there are several Kubernetes clusters that interact with each other. For the sake of separability and compartmentalization, it is best to use a single tenant for monitoring multiple Kubernetes clusters. In this way, the OpenTelemetry Collectors on all of the nodes in all of the clusters used in production can push their collected data to this one monitoring backend. All the monitoring data is thus aggregated in one centralized place for developers to easily access and interpret.

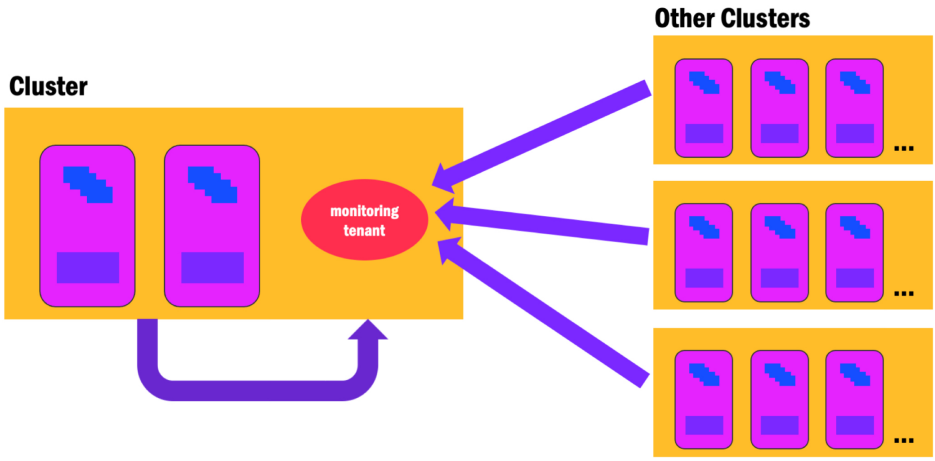


Figure 2.2: The relationship between Kubernetes clusters

After setting up the OpenTelemetry Collector, we then have to set up a custom receiver for our backend system to process the exported metrics. Since OTLP already provides a standardized shape for telemetry data, we decided to set up an HTTP endpoint that receives OTLP data. Using OpenTelemetry's Java library, we are able to parse the metric data into details to reformat into the existing backend's expected shape.

The final end-to-end pipeline for the metrics component of our observability framework is below in Figure 2.3 as follows:

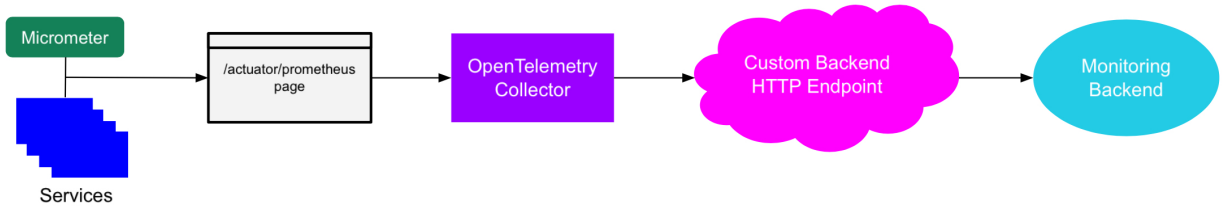


Figure 2.3: Flow chart for metrics

### 2.2.2 Customizable Switches

We wanted to give the option to easily remove metrics data, should developers not want to use it for some reason. To do this, we added in customizable switches to turn off exporting

metrics. The Collector obtains metric data by scraping from the `/actuator/prometheus` endpoint of a given service, but we can stop this export of metric data by removing the endpoint as a whole. Despite having the `/health`, `/info`, and `/prometheus` endpoints enabled by default, each service can still customize which `/actuator` endpoints it exposes, so by overriding these endpoints to exclude the `/prometheus` endpoint, a service can remove its metrics from detection by the Collector. These endpoints can be overridden by either by setting the `management.endpoints.web.exposure.include` property in an `application.properties` file within the service or by changing an environment variable on the Kubernetes pod.

Furthermore, each service can selectively choose to export (or not export) certain groups of metrics. We included this feature in case there were any automatically instrumented sets of metrics that were definitively unhelpful to have for a given service. Micrometer has a `MeterFilter` object [5] that can be used to create rules around which metrics to include and to transform these metrics as well. In order to allow developers to configure which metrics to export, we created a Spring Bean [16] of a `MeterFilter` that would take in a comma-delimited string environment variable, which is something easily changed at any point in time by a developer, and filter out any metric that had a name starting with one of the arguments. For example, if we wanted to filter out metrics on the Java virtual machine and log4j2 logger, we would set `"jvm,log4j2"` as the environment variable. The `MeterFilter` is specifically configured for the `PrometheusMeterRegistry`, which is the `MeterRegistry` for the Prometheus backend service. Since we are using the Collector to scrape from the `/actuator/prometheus` endpoint, this `MeterFilter` works as we would like by filtering out the specified metric groups from the `/actuator/prometheus` endpoint such that the Collector never scrapes them.

See Appendix B for the code to implement filters on different groups of metrics.

### 2.2.3 Custom Metrics

Despite Micrometer providing the majority of helpful observability metrics out-of-the-box where there were none before, services can sometimes have service-specific needs for metrics that would be very helpful to see. As part of our observability framework, we wanted to make sure there would be a standardized method for developers to configure any additional custom metrics.

The first major consideration was how to represent all the different `Meter` primitives within our system. We wanted to be able to edit and customize the metrics within our system, but we were not able to directly edit the `Meter` source code from Micrometer, so we decided to make a wrapper parent class for `Meter`, with individual wrapper classes inheriting from this parent class for each `Meter` primitive, i.e. `Timer`, `Counter`, `Gauge`, etc. Having these wrapper classes thus allows us and future developers to easily make changes to all custom metrics within the microservice architecture at once.

The second major consideration was how to address naming these custom metrics. There should be a standardized naming convention because custom metrics must be specially configured, so they should have a higher degree of importance attributed to them. As such, we decided the best and most straightforward method would be to prefix all custom metrics with `"custom."` automatically. This is done in the `Meter` parent class wrapper, such that future developers do not need to consider it when they are adding in new custom metrics.

### 2.2.4 Dynamic Tags

The most challenging feature to implement was dynamic tags. Multiple tags, each with a key and value, are associated with each `Meter`. The primary reason for implementing dynamic tags is to incorporate tenant IDs as a tag on each `Meter` because the services are multi-tenant, meaning that multiple tenants (users) use the same resource. Having tenant ID included on the metrics would then provide a much-needed layer of insight into which tenant a given

metric applies to, instead of having a blanket metric compiling the data across all users. In cases like HTTP request metrics, it is not very helpful to see these aggregated metrics. For example, if a user is having trouble with accessing a certain HTTP endpoint, the aggregate metrics would not tell the developer much helpful information, but tenant-specific metrics would allow the developer to see which endpoint and which users are being affected so that the developer can proactively fix the bug. Since `Meter`s with the same tags are automatically combined, we wanted to make it so that each metric is updated based on which tenant is using the resource, which also means that the tags cannot be statically set ahead of time because they change dynamically based on the current tenant.

Micrometer does not currently have an out-of-the-box solution for dynamic tags. While it initially seems like it would be a relatively easy feature to implement by converting the static values for each tag to dynamic variables, the way in which Micrometer represents its `Meters` prevents this feature change. Within the meter registry, any two `Meters` are considered the same if they are the same type and have the same name and tags. With dynamic tags, two `Meters` could then suddenly become one when a dynamic tag value collides with a static tag value on another meter [17].

Implementing a custom solution for dynamic tags proved to be complicated as well because each `Meter` does not always maintain a reference to the meter registry that it is attached to, so branching off a new metric every time a tag value changes would not necessarily work because it would not be added automatically to the original `Meter's` registry, in which case it would not show up anywhere. Furthermore, for certain types of `Meters` like `Timers`, all of the options used to create the original `Meter` are not preserved after it is created, so any newly created `Meters` of these types would be missing information [18].

Given the challenges with implementing a generic dynamic tags feature that could be applied to any `Meter`, we decided to narrow the scope of the feature to more directly address the problem at hand: seeing tenant IDs on HTTP metrics. Once we did this, it became a much easier problem to solve. Spring Boot 3, the most recent version of Spring Boot as of the

time of this paper's writing, includes a new component for handling HTTP requests called `ServerRequestObservationConvention`. The default HTTP metrics that Micrometer provides [4] includes tags for request method type (e.g. `GET`), the exception class name if any exception is thrown (e.g. `NullPointerException`), HTTP status codes (e.g. `404`), outcome (e.g. `success`), and Uniform Resource Identifier (URI) template (e.g. `/api/user/{id}`). This new `ServerRequestObservationConvention` component now allows us to add on to this list of default tags by overriding the `DefaultServerRequestObservationConvention's getLowCardinalityKeyValues()` method, which is only invoked for metrics (in contrast, the `getHighCardinalityKeyValues()` method is only invoked for traces) [19].

See Appendix C for the code to implement dynamic tags for tenant IDs on HTTP request metrics. When overriding the `getLowCardinalityKeyValues()` method, we add in another method `additionalTags()`, which checks for the presence of custom header `"X-Tenant-ID"`, and adds it in as a tag if so. If the header is not present, we set this tag value to `"null"`. This is important because Micrometer cannot handle having two metrics that are nearly identical (in type, name, tags, etc.) except with one having an additional tag as well. Micrometer will only display one of these almost-identical metrics, and it will choose to display whichever one is accessed first. Without this `"null"` default value then, all the desired metrics will not show up. This workaround also makes sense from a logical perspective though because all metrics without a tenant ID associated with it should be grouped together if they are the same in every other way. They can also be filtered on having a `"null"` tag value, whereas they would not be able to be filtered out otherwise.

Even though we were not able to fully configure dynamic tags on all metrics, we are at least able to configure dynamic tags for the specific metrics that matter most to have them.

# Chapter 3

# Logs and Traces

Logs and traces are the two major components of observability aside from metrics. To reiterate, logs record events and can show users more information about these events, such as which ones occurred and when they occurred. They can provide any information that a developer wishes to know about the state of the system at the time the event occurred, as long as there is a way to access that state through code. Traces extend the idea of logs by building in a causal ordering of events, helping users trace the outcome of an error back to the error itself. In a microservice architecture, this requires communication and coordination between distributed services. Traces provide a sequence to the events described by logs. This assists developers in gaining a better idea of what happened as a whole and how the different events are interconnected.

Metrics, logs, and traces are all important in observability, and these three components ultimately form the observability framework that we are establishing in our decentralized system.

## 3.1 Logs

There are logs emitted for every kind of event. As such, it is easy to get bogged down in a flood of logs, making it difficult for developers to parse through them all to find helpful

information. At a certain point, they are more helpful to not have rather than to have. This balance of having too much information versus too little information can be mitigated if there is an effective way to organize the logs and sift through them easily.

In the rest of this section, we describe a system that allows developers to utilize all of the information provided by logs efficiently in their development by using a standardized log message structure and sectioning out each category of information, thereby helping developers debug faster with only the most relevant pieces of information. To return to our running example of the developer fixing the user's slow requests issue, having organized logs would have allowed the developer to filter log messages from a specific service with a certain log level (e.g. FATAL, ERROR, WARN), thereby removing the need for them to parse though hundreds, if not thousands, of log messages. Once the developer identifies some problematic log messages, they can then narrow down the sections of code that emitted these log events to focus on only those snippets.

### 3.1.1 OpenTelemetry Collector

We first return to the OpenTelemetry Collector. There are a couple ways to install the OpenTelemetry Collector. For metrics, we used the OpenTelemetry Kubernetes Operator, which automatically installs the Collector into a Kubernetes system and actively monitors its instances. The Kubernetes Operator is the fastest method because it abstracts out much of the configuration work. However, another method that provides more customizability is to install the Collector via a Helm chart [20].

A Helm chart is a collection of files that create a set of Kubernetes resources. It can be very simple, creating only a single Kubernetes pod, or it can be very complex, with a hierarchical directory structure and interconnected custom resources. OpenTelemetry provides a Helm chart to install the Collector, and this Helm chart can be easily edited by developers to completely customize the Collector for the developer's purposes [21].

OpenTelemetry's Collector Helm chart comes with a set of configurable presets for a

few common tasks that require complex alterations to the Kubernetes deployment of the Collector. Enabling any of these presets abstracts away these complicated steps by triggering the associated Helm charts when used. The preset that is particularly helpful to us here is the `logsCollection` preset. One limitation of using presets, however, is that they are not easily configurable. In the future, we hope to rectify this problem. More on that in chapter 5.

Without the preset, our Kubernetes containers send all of their logs to standard output. Currently, our system uses FluentBit, which scrapes these Kubernetes logs from standard output [22]. FluentBit is another Cloud Native Computing Foundation (CNCF) open-source project, designed to integrate well with other CNCF projects such as Kubernetes and OpenTelemetry. Its focus is to streamline performance and consume fewer resources. Like the OpenTelemetry Collector, it is used to manage telemetry data from various sources and process this data in one place. However, since we are already using the OpenTelemetry Collector for metrics purposes, we wanted to explore whether we would be able to process log messages through the Collector as well, instead of using FluentBit. Enabling this functionality would open the door to potentially removing the need for FluentBit and consolidating the technical stack by one less product.

Similar to how FluentBit scrapes the Kubernetes logs from standard output, we can configure the OpenTelemetry Collector to scrape logs from standard output as well. Part of what the `logsCollection` preset does to handle this is to configure a `Filelog` receiver, which is a type of receiver on the Collector that scrapes and parses logs from files. The `Filelog` receiver can only collect logs on the specific Kubernetes node that the Collector is running on, and having multiple Collectors on the same node produces duplicate log data, so it is recommended that this preset be used with Collectors being run in the Kubernetes daemonset mode. The daemonset structure means that there is exactly one Collector running on each node, as seen in Figure 2.1. We are already running the Collector in the daemonset mode for metrics, so this aligns well with our pre-existing Collector setup.

Now that we have configured the Collector to scrape the logs from the Kubernetes pods'

standard output files, we then need to ensure we are parsing these logs correctly. We had to make numerous tweaks to the `logsCollection` preset's parsers because the structure of our logs did not line up with what the parsers expected. These changes can be made by overriding the `Filelog` receiver automatically instrumented by the preset and updating the operators involved in the receiving of the data to expect the correct form of log messages. For the configuration we used, see Appendix D. Developers may customize this configuration to their specifications.

### 3.1.2 Logs Ingestion Endpoint

At this point, we have successfully scraped the logs and parsed out the information required to achieve the OTLP specification for logs [7]. Our next and final step is to connect these logs, being piped through the Collector, to our monitoring backend.

We modeled our approach to connect logs to our monitoring backend off of our previous approach to connect metrics to our monitoring backend. In line with the principles of using a microservice architecture, we set up a new HTTP endpoint in a separate service for ingesting and managing log messages. This new HTTP endpoint expects to receive log data in accordance with the OTLP specifications and uses OpenTelemetry's Java library to reformat the data into the existing back end's expected shape. Future developers can choose to configure various transformations and filters for log messages in this layer (in addition to the operators in the Collector) if they so choose.

With the connection to the backend implemented, the log messages from Kubernetes pods are now able to be scraped into the Collector and exported to the monitoring backend system, where the logs can be filtered and sorted by different parts of the message according to what is most relevant and useful to the developer.

The final end-to-end pipeline for the logs component of our observability framework is below in Figure 3.1 as follows:

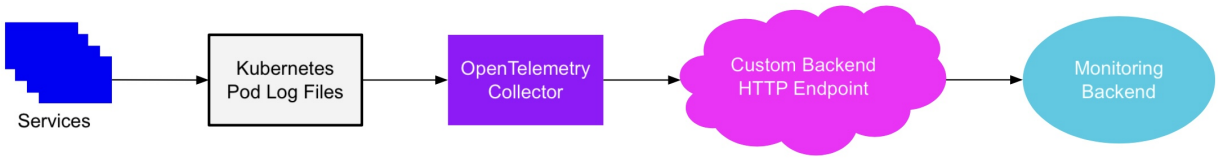


Figure 3.1: Flow chart for logs

## 3.2 Traces

Traces are perhaps the most complex element of observability. They involve many moving parts, whereas metrics only need to track one measure at a time, and logs essentially are print statements triggered when their code is accessed. In contrast, since traces provide a causal ordering of events, they must span different types of events and record which event leads to which other event. This becomes even more difficult in a microservice architecture, where calls span different services, each with its own purpose. This is where distributed tracing comes in.

### 3.2.1 Distributed Tracing

Distributed tracing is a method of tracking application requests across services and databases, with many spans comprising a single trace. It is especially important and relevant in a microservice architecture, where there are many interconnected services constantly communicating with each other to serve a single user request. Figure 3.2 gives an example of a distributed trace.

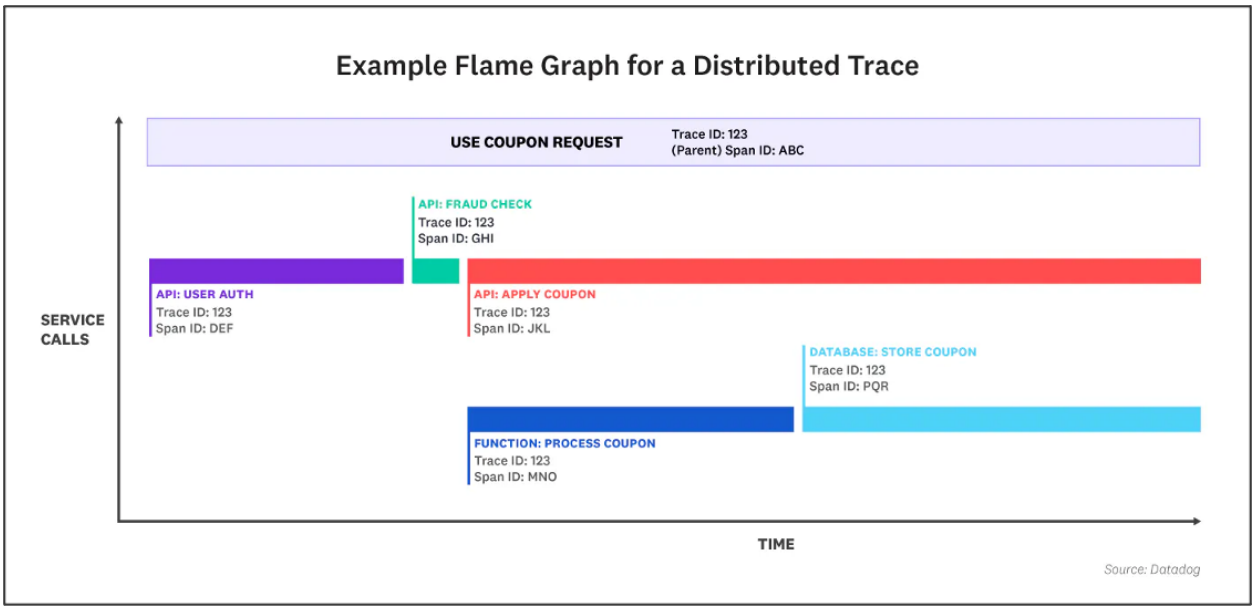


Figure 3.2: Distributed trace example, provided by Datadog [23]

The figure above shows how a trace can span many different types of events, from API and function calls to database accesses to user requests. Throughout the trace, the trace ID remains constant, but each event has a different span ID, representing the entire action conducted in the event. Many spans comprise a single trace. For example, in the figure, the dark blue bar represents the length of time that it took the system to execute the `"PROCESS COUPON"` function call. We also can see that this function call was triggered by the green `"FRAUD CHECK"` API call because the dark blue bar started exactly where the green bar ended, and the `"PROCESS COUPON"` function call subsequently triggered the light blue `"STORE COUPON"` write to database because the dark blue bar ends exactly where the light blue bar begins.

As evidenced by the different levels of bars in the graph, a single request can trigger multiple actions. This is part of what makes distributed tracing so difficult; there can be a large number of spans in one trace within a distributed system if the services within the system are highly interconnected. Each span must always keep track of who its parent is, i.e. which action/span triggered it.

The graph provides a very simple example of all the events that can be triggered by

a single user request. In a more complex and connected system, there can be many more services and components involved in a single user request. It can thus be very complicated for a developer to debug a single issue because the issue could have originated in many different places. Having distributed traces would help the developer tremendously by narrowing down the specific chain of events that led to the issue, so the developer needs only to examine each event in the chain to pinpoint where the error occurred. For example, if the developer from our previous scenario involving a user's slow HTTP requests had access to distributed traces, they would be able to narrow down the components they look at to be only the ones that the user's request touched, which would be easily accessible by simply following the trace for that one request. Seeing the time spent to execute each call further allows the developer to tunnel down into which specific component in the flow is contributing the most to the user's high latency issue. It is in this way that traces help developers to determine the origin of errors, and because they include the timing of each call, they are particularly helpful for high latency errors.

### 3.2.2 OpenTelemetry Java Agent

As we already instrumented OpenTelemetry components through much of the system for metrics and logs, it was natural to continue using OpenTelemetry for traces. OpenTelemetry has a Java agent that provides automatic instrumentation for metrics, logs, and traces.

A Java agent is a specific type of file that contains compiled Java code and uses the Java Instrumentation API that the Java Virtual Machine (JVM) provides to alter the low-level code that is loaded into the JVM [24]. This means that the Java agent is run every time Java code is run. A Java agent can be loaded in either statically, by adding in a `-javaagent` flag whenever an application is started up, or dynamically, by attaching the agent to code that is already running via the Java Attach API.

The OpenTelemetry Java agent can automatically instrument traces into a distributed system. It does this by injecting byte-code into the JVM that creates a trace for each request

and corresponding spans to follow the actions carried out by the request through the entire microservice system. By using the OpenTelemetry Java agent and exporting the traces, we are thereby able to configure distributed traces into our system. From there, we can export these traces in OTLP format to the Collector, which will parse them into trace objects that include the trace ID and span ID, allowing for distributed traces to be correlated.

The existing system had no support for traces, while it did already have some support for metrics and logs, so there is still much work to be done on this front. This work includes creating a new service specifically for traces and processing the traces exported to the Collector. The new service then needs to have an HTTP endpoint to receive OTLP-formatted traces and export them to the monitoring backend visualization platform.

When the future work on traces is completed, the final end-to-end pipeline for the traces component of our observability framework should look like Figure 3.3:

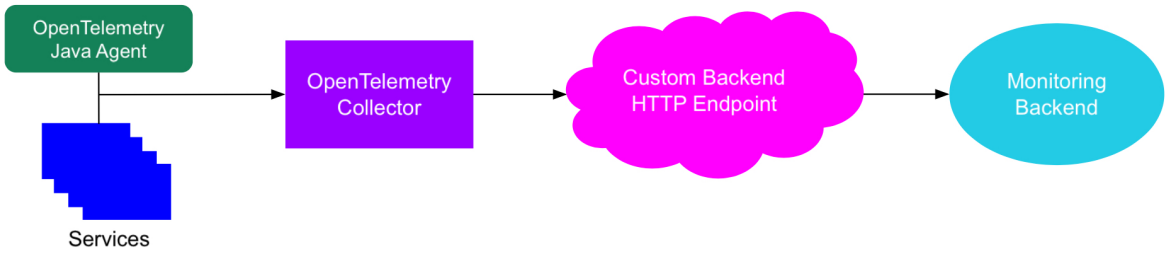


Figure 3.3: Flow chart for traces

More on this in chapter 5.

# Chapter 4

# Results

Much of the data that exists in today's systems within industry settings are not helpful to developers and end up introducing a lot of noise into the system as a result. The major goals of this research were to successfully instrument a new framework for observability into a distributed, microservice architecture, and to have this framework provide the information that developers needed to effectively monitor large systems while still being highly customizable for future developers to iterate upon. In the next few sections, we review the results of implementing everything we discussed in the previous chapters.

## 4.1 State of Metrics

While there are many metrics that provide information that already exist in systems, they often have many limitations: they may be incomplete if there are not many of them; they may lack granularity if they do not exist by service; and they may provide insufficient visibility into the system if they do not contain any application performance metrics (e.g. metrics on the web server, memory usage metrics). However, after building in end-to-end compatibility in our system with Micrometer and the OpenTelemetry Collector, each service now has over 50 metrics available that exist specifically for that service. It is likely that a service has more than 50 metrics available too depending on which technologies they use, as Micrometer

provides metrics by default for all frameworks that are being used as long as the framework is supported. A service will also have more metrics available if it requires tenant-specific HTTP metrics, as each tenant will have its own set of HTTP metrics then, or if it has many HTTP endpoints enabled, as each endpoint will have its own set of HTTP metrics. On average though, assuming dynamic tags on tenant IDs are not yet enabled and there is only 1 HTTP endpoint on the service, each service now has 60 metrics out-of-the-box, compared to 0 metrics before this framework was implemented.

Figure 4.1 and Figure 4.2 below provide examples of metric graphs available on the monitoring backend after fully configuring Micrometer and the OpenTelemetry Collector. Each metric graph can also be filtered by tags, which is where dynamic tags for tenant IDs on HTTP metrics are very helpful.

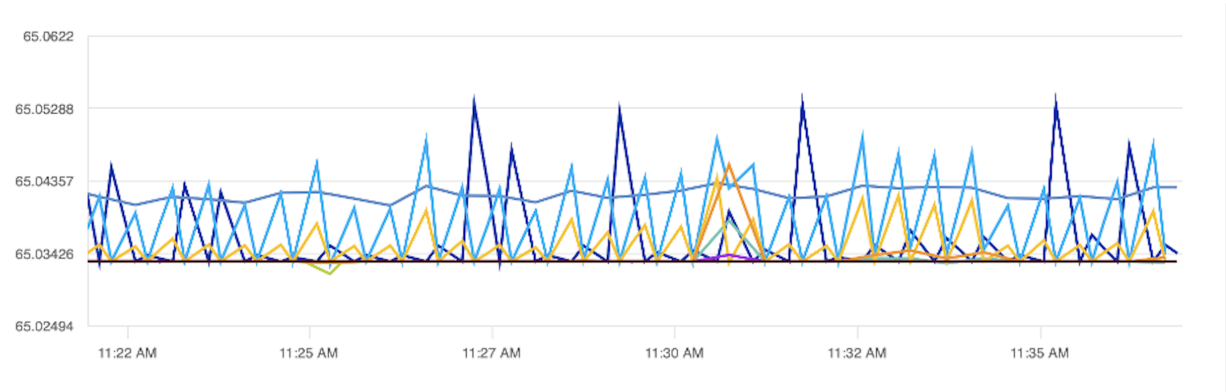


Figure 4.1: Average free bytes available on the disk, by service

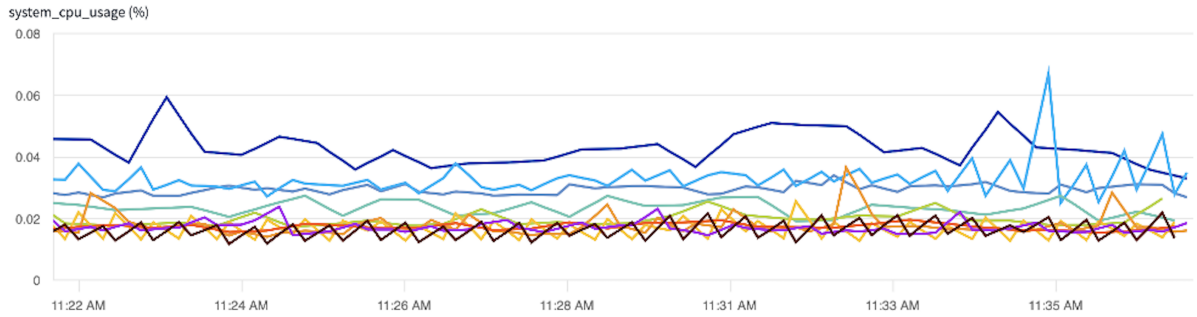


Figure 4.2: Percent of system CPU usage, by service

The graphs in Figure 4.1 and Figure 4.2 only provide examples of 2 out-of-the-box metrics.

However, we want to evaluate how useful these default metrics really are. Interviews with SREs operationally managing SaaS (software-as-a-service) products revealed that it was most important to have metrics that measured the four "golden signals" of monitoring: latency–the time it takes to service a request; traffic–a measure of how much demand is being placed on the system; errors–the rate of requests that fail; and saturation–how "full" the service is. Luckily, we are able to identify default metrics in our framework that satisfy each of these golden signals. Table 4.1 lists which default metrics are helpful in measuring each golden signal.

Table 4.1: Default metrics that help measure the golden signals of monitoring

| Signal | Metric |
|---|---|
| Latency | - HTTP server requests seconds max, separable by status and method (to track error latency as well)<br>- application ready time seconds |
| Traffic | - HTTP server requests seconds count, separable by method |
| Errors | - HTTP server requests seconds count, separable by status (i.e. not 200)<br>- log4j2 events total, separable by level (i.e. error/fatal) |
| Saturation | - system CPU usage<br>- system load average<br>- process CPU usage<br>- JVM buffer memory used, in bytes<br>- JVM garbage collection overhead percent<br>- JVM memory usage after garbage collection percent<br>- JVM memory used, in bytes<br>- JVM memory max, in bytes<br>- executor pool size threads<br>- executor pool max threads<br>- disk free bytes<br>- disk total bytes |

Now that we have seen that the default metrics of our framework satisfy SRE requirements, let us look at the custom metrics part of the framework. There are 2 parts to creating a custom metric: the measurement-specific portion of writing functions to take these measurements and the integration-specific portion of incorporating these measurements as a new

metric in the meter registry. The former part can vary greatly in complexity and effort, based off of what is being measured, but the latter part is within our control, so we will focus on that. With the new framework, the integration portion of creating a custom metric requires only one line, for example:

```
List<String> customMetric = (new CustomGauge<>(meterRegistry, "measurements",
new ArrayList<>(), List::size)).getObj();
```

With this one line, we successfully instrument a custom `Gauge` object with a name that automatically begins with `"custom."`, as our framework has dictated. Any further standardizations we want to apply to all of our custom metrics can be changed in the `CustomMetric` wrapper class, instead of needing to edit each individual custom metric, as was the case previously.

Since this framework has not been fully rolled out, we cannot say for certain how many custom metrics will be required given that it seems like the default metrics satisfy requirements so far, but we will approximate that custom metrics will comprise 10% of all metrics, with the remaining 90% being the default metrics. With the declaration of a custom metric requiring only 1 line of code then, the number of additional lines of code to create a custom metric is minimal. Operating under the assumption of custom metrics comprising 10% of all metrics, and using the previous measurement of 60 default metrics on average added in each service, we can estimate the total number of additional lines of code needed in our framework to be approximately 6 if all the functions required to perform the metric's measurements are already present. As this is a relatively small number, and it still ensures standardization across all custom metrics, we consider our framework to be successful on the custom metrics front.

The main benefit of instrumenting metrics in this observability framework is to shift the approach towards system monitoring to be more proactive rather than reactive. Since this research is targeted towards industry uses, it is important to note that oftentimes in industry settings, developers only fix problems once an issue is propagated through the system and

results in an error. However, by monitoring multiple metrics across all parts of a system, we are able to create a baseline measure of what to expect in a correctly operating system, so we can be more proactive in identifying and resolving potential sources of issues by seeing when a metric deviates from what we would expect instead of waiting until errors are reported.

## 4.2 State of Logs

Figure 4.3 below provides an example of a logs graph available on the monitoring backend after configuring the `logsCollection` preset on the Collector and the logs ingestion HTTP endpoint on the monitoring backend service. Users are able to filter by the log message's severity level (to filter out the more important messages) and source as well. They can also sort the messages by time or select a specific frame of time to evaluate, thus enabling them to ignore any logs that are not relevant to their work.

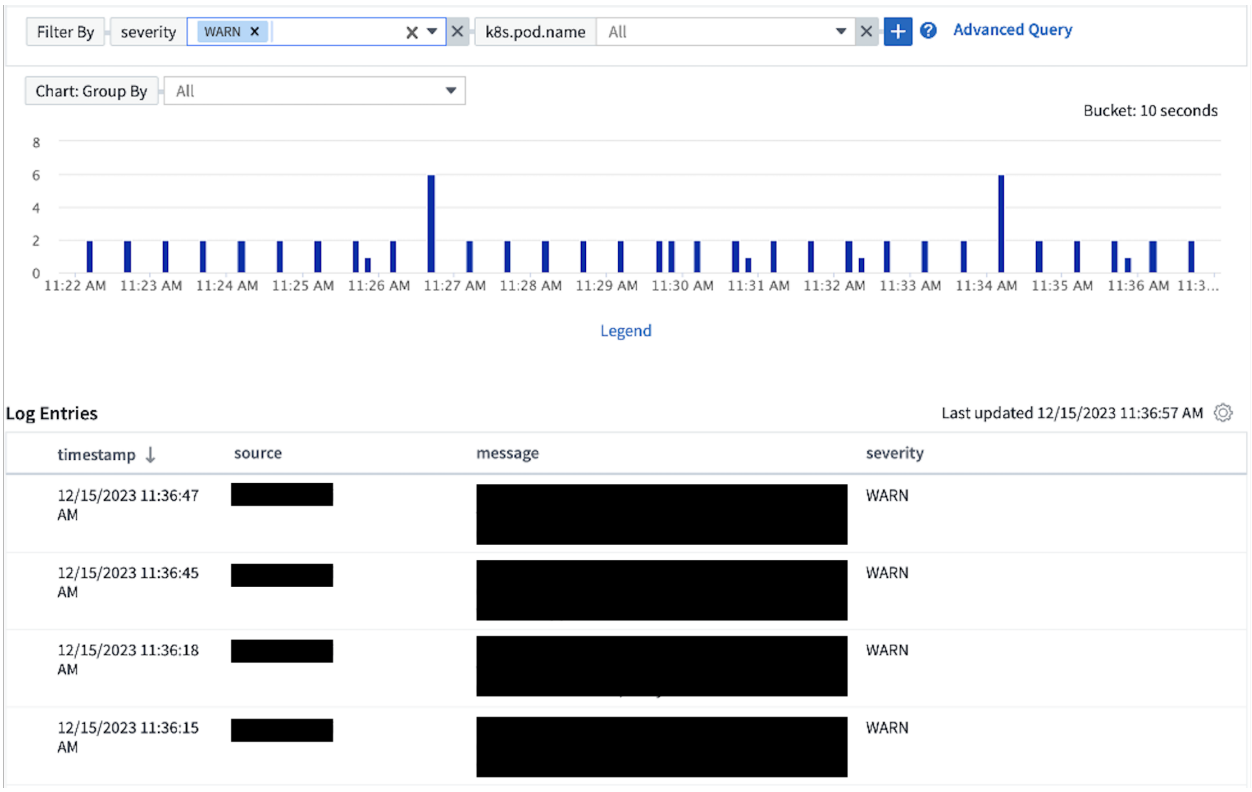


Figure 4.3: Log messages by time, source (which service it originated in), message text, and severity level

## 4.3 State of Traces

Traces are a little more complicated because our framework has not provided a comprehensive solution for them. Instead, we mainly gained a deeper understanding of what was possible with automatic instrumentation and what we may still want to manually instrument: the OpenTelemetry Java agent supports REST endpoint accesses and database accesses, so the bulk of what developers want to see is already present. Any additional manual instrumentation would be minimal.

## 4.4 Overall Impact

Our framework instruments observability data for metrics, logs, and traces, which results in increased amounts of data for our system. While we cannot publicly reveal the exact amount of load our system can handle, we can provide an equation that approximates the number of data points emitted per second, including the estimated number of custom metrics. The following equation uses $s$ to represent the number of services in the microservice architecture and $p$ to represent the number of pods (replicas) per service. By default, the OpenTelemetry Collector pushes data once every 60 seconds, but this frequency is configurable.

$$\begin{aligned} \text{total data points} &= 66 \text{ metrics} * s * p * 1 \text{ push}/60 \text{ seconds} \\ &= 1.1sp \text{ data points pushed per seconds} \end{aligned}$$

The previous equation assumes that dynamic tags are not yet enabled on HTTP metrics, so if we assume that dynamic tags are enabled for tenant IDs on HTTP metrics and we are only using one HTTP endpoint per service, the new equation becomes the following, with $t$

representing the number of tenants for a single resource:

$$\begin{aligned}\text{total data points} &= (65 \text{ metrics} + 1 \text{ HTTP metric} * t) * s * p \\ &\quad * 1 \text{ push}/60 \text{ seconds} \\ &= (65 + t)/60sp \text{ data points pushed per second}\end{aligned}$$

To give an idea of what scale this is on, we can plug in some sample values into this equation. A given system usually has less than 100 services, but for a scaled scenario, we can assume $s = 100$. The number of pods per service is usually 2, but it can go up to 10, so in our scaled scenario, $p = 10$. Finally, the number of tenants for a given resource can fluctuate depending on the system, but we will assume that large systems have 500 tenants, and the average system has 200 tenants, so for the scaled scenario, we will assume $t = 500$.

$$\begin{aligned}(65 + t)/60sp &= (65 + 500 \text{ tenants})/60 * 100 \text{ services} * 10 \text{ pods/service} \\ &= 9,417 \text{ total data points per second}\end{aligned}$$

This is a scaled scenario. To give an example of an average scenario, let us assume $s = 50$, $p = 2$, and $t = 200$:

$$\begin{aligned}(65 + t)/60sp &= (65 + 200 \text{ tenants})/60 * 50 \text{ services} * 2 \text{ pods/service} \\ &= 442 \text{ total data points per second}\end{aligned}$$

Given that all of these values can vary from system to system and depend on specific configuration details of the OpenTelemetry Collector, these numbers are only provided to give an approximate idea of how many data points are being exported by our framework.

With an observability framework that generates metrics and traces and consolidates them with logs in one common Collector, SREs and developers now have access to much more information on the state of their systems than they did before, and they have many

more tools at their disposal to help them debug issues as well. Users benefit from this framework too because they now have the ability to visualize OTLP-formatted data from any source. Since OpenTelemetry is the emerging industry standard, this means that there are many more products that can be integrated with this monitoring backend, making it a more comprehensive platform overall. For example, Amazon Web Services' (AWS) CloudWatch [25], which is a separate, open-source observability platform, can export observability data in OTLP format, so users who employ multiple observability solutions can now consolidate their data in one complete monitoring backend platform.

# Chapter 5

# Other Considerations and Future Work

Observability is a continuously developing field as the systems we monitor continue to develop and change—the tools we use to monitor these systems must then develop and change as well. There is always more to do to monitor a system. With this in mind, here are a few ideas for future work on developing out this observability framework.

## 5.1 Future Work on Metrics

The following are features that may be added to make our proposed framework further extensible and customizable:

- Extend the custom metric wrapper classes to all `Meter` primitives, since only 4 are represented right now: `Counter`, `Timer`, and `Gauge`. Unrepresented `Meter` primitives include `DistributionSummary`, `LongTaskTimer`, `FunctionCounter`, `FunctionTimer`, and `TimeGauge`.

- Configure support for users with data already in OTLP format to use the metrics ingestion HTTP endpoint. This work should be minimal, primarily ensuring the data that users are exporting are encoded and formatted properly, since the endpoint already ingests the OTLP format.

## 5.2 Future Work on Logs

There is still more that can be done with logs to clean up the organization of the code:

- Implement the `logsCollection` preset manually to reduce redundancy when customizing the component. Currently we are using the preset to establish a baseline setup, then we override that with various customizations to the `Filelog` receiver of the OpenTelemetry Collector, but if we configure the collection of log data manually instead of using the preset, we will not need to override anything in order to customize the Collector.

- Extract more of the parsing and processing of the logs into the services to keep the Collector more lightweight and scalable. There may be some experimentation required in finding the perfect balance.

## 5.3 Future Work on Traces

Future work on traces can be broken down into the following steps:

1. Decide whether any manual instrumentation is necessary. If so, OpenTelemetry provides a Java SDK (software development kit) that is very helpful in configuring distributed tracing. It will automatically instrument anything that it can first, along with anything the developers build in. The benefit of manual instrumentation is that the observability system can focus on capturing custom metadata in specific areas of interest.

2. Create an HTTP endpoint that can receive traces in OTLP format, parse them, and represent them in the custom backend.

3. Correlate trace IDs to logs by adding attributes into logs in order to see which logs correspond with a given trace, thereby further filtering out extraneous log messages.

Once traces can be processed and ingested into the monitoring backend, the overall flow of observability data will look like the diagram below in Figure 5.1.

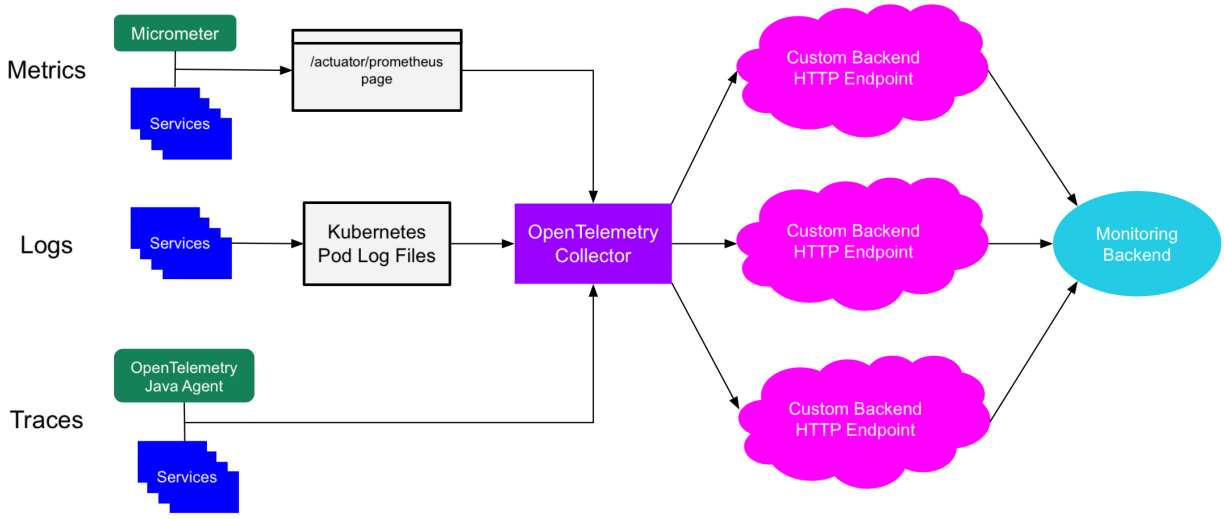


Figure 5.1: Flow chart for metrics, logs, and traces

## 5.4 Other Considerations

There are a few other considerations to have moving forward, beyond the future work that needs to be done. Some of these considerations pertain to monitoring ongoing developments in the observability space, and others pertain to tradeoffs that may occur in certain design decisions.

With metrics, we had limited the scope of the goal for dynamic tags to only apply to HTTP metrics, but ideally, we would be able to implement dynamic tags for any type of metric. Since this problem is mainly solvable by the Micrometer developers, it is important to monitor future iterations of Micrometer to see if any updates make dynamic tags more feasible to implement.

With logs, we must weigh the tradeoffs of cutting out FluentBit in favor of the OpenTelemetry Collector. There is still much to be done in terms of seeing how much is stream-

lined from having all the observability components piped through the same technology (the OpenTelemetry Collector) as opposed to having them all be processed separately in different technologies. If there is a significant performance impact, it may be worth considering reinstating FluentBit, as it is specifically designed for streamlining performance and consuming fewer resources.

With traces, one technology that we explored was eBPF tracing [26]. eBPF, which originally stood for extended Berkeley Packet Filter but is now considered a standalone term, allows programs to run within an operating system, so developers can add functionality to the operating system at runtime. When extending this technology to tracing, it means that tracing is lightweight, scalable, and fast, as eBPF programs track instructions as they execute. Since eBPF operates at the level of the kernel, it cannot access native Kubernetes data; however, it is able to get information (such as IP address, source, and destination) that can then be correlated with Kubernetes pods. There are two main approaches to implementing an eBPF program: kernel probes (kprobes) and user probes (uprobes). Kernel probes have security and privacy concerns, as they do not work with Transport Layer Security (TLS), while user probes have scalability concerns, as they need to be implemented for each client library being used and must be updated every time a library is updated.

OpenTelemetry has an eBPF Collector, but it is still being updated frequently. Given all of these concerns, we decided against using eBPF tracing, as it is still a rapidly developing area, and implementing it would make the entire microservice system subject to frequent updates and likely more frequent breakages too. However, it is certainly something to consider for the future, once updates stabilize and security tightens.

# Conclusion

This thesis describes an observability framework incorporating the three key elements of observability: metrics, which are data collected from measuring events; logs, which record events and provide more information and context about them; and traces, which provide a sequential ordering of events. This framework is relevant to every company in the industry because observability plays a key role in keeping systems up and running.

We developed this framework originally with the design goals of flexibility and customizability in mind, and we have achieved these goals. By ensuring the framework does not need to be implemented in every microservice in order to be utilized, but still allowing for microservices to override generalized features, we create a balance between a scalable system and a flexible one. By laying out a framework for custom metrics and using technologies with modifiable configurations like the OpenTelemetry Collector, we ensure that our system is customizable.

Prior to the work described in this thesis, we have seen that many systems do not have any sufficient solution to the issue of observability, much less observability in a microservice architecture. Some had very few metrics, with even less of these metrics being helpful towards monitoring an application's performance, and there was no organized way to parse out valuable data in log messages, along with no distributed traces whatsoever.

The observability framework that we laid out here integrates seamlessly into the existing services and backend monitoring platform. The many features of our framework are as follows:

- All of the services within the microservice architecture are able to reflect the same changes by making the changes in a base library that all of the services inherit from.
- Any service can choose to stop exporting observability data points by simply switching off that feature using a specific application property, and any service can similarly choose to stop exporting certain groups of observability metrics by specifying them in a different application property.
- Should the default metrics provided by Micrometer lack in anything, developers can create their own custom metrics in a standardized manner using the custom metric wrapper classes.
- Tenant IDs are also included in HTTP metrics to identify more easily which tenants are being affected by a given issue.
- Metric graphs are available in the monitoring platform, separable by a metric's tags or any other attribute.
- Log messages can be processed in many different ways through either the OpenTelemetry Collector or in the microservices.
- Log graphs are also available in the backend monitoring platform, separable by attributes such as timestamp, source, and severity level.
- Distributed traces are automatically instrumented throughout the services by using OpenTelemetry's Java agent.

There is still much work to be done on this framework, as evidenced by chapter 5, but it has great potential to help SREs manage and streamline system observability within our microservices. In this way, we can reduce the time both SREs and developers need to spend on identifying and debugging issues, and we can increase efficiency as a whole.

# Appendix A

# OpenTelemetry Collector Configuration

This is a sample configuration for the OpenTelemetry Collector that receives telemetry data via OTLP, processes the data in batches, and exports the data via OTLP.

```
receivers:
  otlp:
    protocols:
      http:
        endpoint: 0.0.0.0:4318
processors:
  batch:

exporters:
  otlp:
    endpoint: otelcol:4317

service:
  pipelines:
    traces:
      receivers: [otlp]
```

```
      processors: [batch]
      exporters: [otlp]
    metrics:
      receivers: [otlp]
      processors: [batch]
      exporters: [otlp]
    logs:
      receivers: [otlp]
      processors: [batch]
      exporters: [otlp]
```

# Appendix B

# Code for Metric Filters

This is a Spring Boot configuration class that customizes our Micrometer meter registry configuration. With the addition of this configuration class, users can change the values of an environment variable during runtime to filter out which metrics are exported based on the prefix of the metric (determined by what the metric measures), delimited by commas. For example, using a value of `"jvm,log4j2"` for the environment variable filters out all JVM and log4j2 logger metrics from the registry, so they will not be exported to the monitoring backend.

```
1  import io.micrometer.core.instrument.config.MeterFilter;
2  import io.micrometer.prometheus.PrometheusMeterRegistry;
3  import org.springframework.context.annotation.Bean;
4  import org.springframework.context.annotation.Configuration;
5  import org.springframework.boot.actuate.autoconfigure.metrics
6  .MeterRegistryCustomizer;
7  import java.util.Arrays;
8
9  @Configuration
10 public class MeterRegistryConfig {
11     @Bean
```

```
12      public MeterRegistryCustomizer<PrometheusMeterRegistry>
            configurePrometheusExport(String prometheusMetrics) {
13          if (prometheusMetrics.length() > 0) {
14              return (registry) -> registry.config().meterFilter(
                    MeterFilter.deny((id) ->
15                      Arrays.stream(prometheusMetrics.split(",")).anyMatch(
                            m -> id.getName().startsWith(m))));
16          }
17          return (registry) -> {};
18      }
19  }
```

# Appendix C

# Code for Dynamic Tenant Tags for HTTP Metrics

This is a custom-built class that extends a Spring Boot 3 component in order to allow HTTP metrics to have dynamic tags. Our example uses a tenant's ID, contained in the header of an HTTP request, as the dynamic tag value.

```
1  import io.micrometer.common.KeyValues;
2  import org.springframework.http.server.observation
3  .DefaultServerRequestObservationConvention;
4  import org.springframework.http.server.observation
5  .ServerRequestObservationContext;
6  import org.springframework.stereotype.Component;
7
8  @Component
9  public class CustomServerRequestObservationConvention extends
       DefaultServerRequestObservationConvention {
10     @Override
11     public KeyValues getLowCardinalityKeyValues(
           ServerRequestObservationContext context) {
```

```
12          return super.getLowCardinalityKeyValues(context).and(
                additionalTags(context));
13      }
14
15      protected KeyValues additionalTags(
            ServerRequestObservationContext context) {
16          KeyValues keyValues = KeyValues.empty();
17          String headerString = context.getCarrier().getHeader("X-
                Tenant-ID");
18          if (headerString == null) {
19              headerString = "null";
20          }
21          keyValues = keyValues.and("custom.header", headerString);
22          return keyValues;
23      }
24  }
```

# Appendix D

# OpenTelemetry Collector Configuration to Parse Logs

This is a sample configuration for the OpenTelemetry Collector, more specifically the operators involved in parsing the logs through the `Filelog` receiver. I have extrapolated parts of the configuration that are not relevant to the operators' configuration with the asterisk ("*") character. Much of this is adapted from the `logsCollection` preset code [27].

```
mode: daemonset

presets:
  logsCollection:
    enabled: true

config:
  receivers:
    filelog:
      *
      operators:
        - type: router
```

```
13          id: get-format
14          routes:
15            - output: parser-docker
16              expr: 'body matches "^\\{"'
17            - output: parser-crio-severity
18              expr: 'body matches "^[^ Z]+ .*(INFO|DEBUG|TRACE|WARN|
                  ERROR|FATAL)"'
19            - output: parser-crio
20              expr: 'body matches "^[^ Z]+ "'
21            - output: parser-containerd
22              expr: 'body matches "^[^ Z]+Z"'
23        - type: regex_parser
24          id: parser-crio-severity
25          regex:
26            '^(?P<time>[^ Z]+) (?P<stream>stdout|stderr) (?P<logtag
                >[^ ]*)
27            .* ?(?P<sev>INFO|DEBUG|TRACE|WARN|ERROR|FATAL)]? ?(?P<log
                >.*)$'
28          timestamp:
29            parse_from: attributes.time
30            layout_type: gotime
31            layout: '2006-01-02T15:04:05.999999999Z07:00'
32          severity:
33            parse_from: attributes.sev
34        - type: remove
35          field: attributes.sev
36        - type: recombine
37          id: crio-severity-recombine
38          output: extract_metadata_from_filepath
```

```
39          combine_field: attributes.log
40          source_identifier: attributes["log.file.path"]
41          is_last_entry: "attributes.logtag == 'F'"
42          combine_with: ""
43          max_log_size: 102400
44        - type: regex_parser
45          id: parser-crio
46          regex: '^(?P<time>[^ Z]+) (?P<stream>stdout|stderr) (?P<
               logtag>[^ ]*) ?(?P<log>.*)$'
47          timestamp:
48            parse_from: attributes.time
49            layout_type: gotime
50            layout: '2006-01-02T15:04:05.999999999Z07:00'
51        - type: recombine
52          id: crio-recombine
53          output: extract_metadata_from_filepath
54          combine_field: attributes.log
55          source_identifier: attributes["log.file.path"]
56          is_last_entry: "attributes.logtag == 'F'"
57          combine_with: ""
58          max_log_size: 102400
59        - type: regex_parser
60          id: parser-containerd
61          regex:
62            '^(?P<time>[^ ^Z]+Z) (?P<stream>stdout|stderr) (?P<logtag
                 >[^ ]*)
63            ?(?P<log>.*)$'
64          timestamp:
65            parse_from: attributes.time
```

```
      layout: '%Y-%m-%dT%H:%M:%S.%LZ'
  - type: recombine
    id: containerd-recombine
    output: extract_metadata_from_filepath
    combine_field: attributes.log
    source_identifier: attributes["log.file.path"]
    is_last_entry: "attributes.logtag == 'F'"
    combine_with: ""
    max_log_size: 102400
  - type: json_parser
    id: parser-docker
    output: extract_metadata_from_filepath
    timestamp:
      parse_from: attributes.time
      layout: '%Y-%m-%dT%H:%M:%S.%LZ'
  - type: regex_parser
    id: extract_metadata_from_filepath
    regex: '^.*\/(?P<namespace>[^_]+)_(?P<pod_name>[^_]+)_(?P<
       uid>[a-f0-9\-]{36})\/(?P<container_name>[^\._]+)\/(?P<
       restart_count>\d+)\.log$'
    parse_from: attributes["log.file.path"]
  - type: move
    from: attributes.stream
    to: attributes["log.iostream"]
  - type: move
    from: attributes.container_name
    to: resource["k8s.container.name"]
  - type: move
    from: attributes.namespace
```

```
          to: resource["k8s.namespace.name"]
        - type: move
          from: attributes.pod_name
          to: resource["k8s.pod.name"]
        - type: move
          from: attributes.restart_count
          to: resource["k8s.container.restart_count"]
        - type: move
          from: attributes.uid
          to: resource["k8s.pod.uid"]
        - type: move
          from: attributes.log
          to: body
        - type: remove
          field: attributes.time
  exporters:
    *
  service:
    *
```

# References


[1] B. H. Miller, H. Liu, and M. Kolle, "Scalable optical manufacture of dynamic structural colour in stretchable materials," *Nature Materials*, vol. 21, no. 9, pp. 1014–1018, 2022, Publisher: Nature Publishing Group, ISSN: 1476-4660. DOI: 10.1038/s41563-022-01318-x. [Online]. Available: https://www.nature.com/articles/s41563-022-01318-x (visited on 05/20/2024).

[2] "What is microservices architecture?" Google Cloud. (2024), [Online]. Available: https://cloud.google.com/learn/what-is-microservices-architecture (visited on 05/20/2024).

[3] "Spring boot," Spring Boot. (2024), [Online]. Available: https://spring.io/projects/spring-boot (visited on 05/21/2024).

[4] "Spring boot actuator," Spring Docs. (2024), [Online]. Available: https://docs.spring.io/spring-boot/docs/current/reference/html/actuator.html (visited on 03/24/2024).

[5] "Micrometer documentation :: Micrometer." (2023), [Online]. Available: https://docs.micrometer.io/micrometer/reference/ (visited on 05/24/2024).

[6] C. N. C. Foundation. "OpenTelemetry collector," OpenTelemetry. (2024), [Online]. Available: https://opentelemetry.io/docs/collector/ (visited on 03/26/2024).

[7] C. N. C. Foundation. "OTLP specification 1.1.0," OpenTelemetry. (2024), [Online]. Available: https://opentelemetry.io/docs/specs/otlp/ (visited on 03/27/2024).

[8] S. Karumuri, F. Solleza, S. Zdonik, and N. Tatbul, "Towards observability data management at scale," *ACM SIGMOD Record*, vol. 49, no. 4, pp. 18–23, 2021, ISSN: 0163-5808. DOI: 10.1145/3456859.3456863. [Online]. Available: https://dl.acm.org/doi/10.1145/3456859.3456863 (visited on 07/10/2024).

[9] S. Niedermaier, F. Koetter, A. Freymann, and S. Wagner, "On observability and monitoring of distributed systems: An industry interview study," in vol. 11895, 2019, pp. 36–52. DOI: 10.1007/978-3-030-33702-5_3. arXiv: 1907.12240[cs]. [Online]. Available: http://arxiv.org/abs/1907.12240 (visited on 07/01/2024).

[10] M. Usman, S. Ferlin, A. Brunstrom, and J. Taheri, "A survey on observability of distributed edge & container-based microservices," *IEEE Access*, vol. 10, pp. 86 904–86 919, 2022, Conference Name: IEEE Access, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2022.3193102. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9837035 (visited on 07/01/2024).

[11] "Multitenant architecture." (2024), [Online]. Available: https://www.cloudflare.com/learning/cloud/what-is-multitenancy/ (visited on 05/24/2024).

[12] "Log4j," Apache Logging. (2024), [Online]. Available: https://logging.apache.org/log4j/2.x/ (visited on 07/06/2024).

[13] "Prometheus overview," Prometheus Docs. (2024), [Online]. Available: https://prometheus.io/docs/introduction/overview/ (visited on 03/24/2024).

[14] C. N. C. Foundation. "Transforming telemetry," OpenTelemetry. (2024), [Online]. Available: https://opentelemetry.io/docs/collector/transforming-telemetry/ (visited on 03/28/2024).

[15] C. N. C. Foundation. "Using OpenTelemetry with kubernetes," OpenTelemetry. (2024), [Online]. Available: https://opentelemetry.io/docs/kubernetes/getting-started/ (visited on 03/28/2024).

[16] "Basic concepts: @bean and @configuration :: Spring framework," Spring Docs. (2024), [Online]. Available: https://docs.spring.io/spring-framework/reference/core/beans/java/basic-concepts.html (visited on 04/01/2024).

[17] J. Schneider. "Micrometer issue #2223 · micrometer-metrics/micrometer," GitHub. (2020), [Online]. Available: https://github.com/micrometer-metrics/micrometer/issues/2223 (visited on 05/27/2024).

[18] J. Schneider. "Micrometer issue #535 · micrometer-metrics/micrometer," GitHub. (2018), [Online]. Available: https://github.com/micrometer-metrics/micrometer/issues/535#issuecomment-433647380 (visited on 05/27/2024).

[19] "Observability support," Spring Framework. (2024), [Online]. Available: https://docs.spring.io/spring-framework/reference/integration/observability.html#observability.config (visited on 05/27/2024).

[20] C. N. C. Foundation. "Helm charts," Helm Docs. (2024), [Online]. Available: https://helm.sh/docs/topics/charts/ (visited on 05/28/2024).

[21] C. N. C. Foundation. "OpenTelemetry collector chart," OpenTelemetry. Section: docs. (2024), [Online]. Available: https://opentelemetry.io/docs/kubernetes/helm/collector/ (visited on 05/28/2024).

[22] C. N. C. Foundation. "What is fluent bit? | 3.0 | fluent bit: Official manual." (2024), [Online]. Available: https://docs.fluentbit.io/manual/about/what-is-fluent-bit (visited on 06/04/2024).

[23] Datadog. "What is distributed tracing? how it works & use cases," Datadog. (2022), [Online]. Available: https://www.datadoghq.com/knowledge-center/distributed-tracing/ (visited on 06/27/2024).

[24] A. Precub. "Guide to java instrumentation," Baeldung. (2024), [Online]. Available: https://www.baeldung.com/java-instrumentation#what-is-a-java-agent (visited on 06/06/2024).

[25] "APM tool - amazon CloudWatch - AWS," Amazon Web Services, Inc. (2024), [Online]. Available: https://aws.amazon.com/cloudwatch/ (visited on 07/02/2024).

[26] "eBPF | an introduction and deep dive into the eBPF technology." (2024), [Online]. Available: https://ebpf.io/what-is-ebpf/ (visited on 06/06/2024).

[27] "logsCollection configuration | opentelemetry-helm-charts," GitHub. (2021), [Online]. Available: https://github.com/open-telemetry/opentelemetry-helm-charts/blob/8319c0fcfdab578f94224fb6b7eefd33944702fa/charts/opentelemetry-collector/templates/_config.tpl (visited on 06/04/2024).