Informatics Institute of Technology

In Collaboration With

University of Westminster, UK

Lazy-Koala: A Lightweight Framework for Root Cause Analysis in Distributed Systems

A dissertation by

Mr. Isala Piyarisi

w1742118 / 2018421

Supervised by

Mr. Guhanathan Poravi

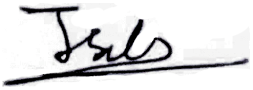
May 2022

Submitted in partial fulfilment of the requirements for the BSc(Hons) Computer Science degree at the University of Westminster.

Declaration

I, Isala Piyarisi, hereby truthfully declare that this project paper is a record of my own and independent research investigation, and any part of this dissertation was not previously submitted/presented to any diploma, degree or other qualification programme. Facts gleaned from reliable outside sources have been appropriately cited and credited.

Student Name - Isala Piyarisi
Registration No. - w1742118 / 2018421



Signature: ..... **Date:** 18/05/2022

Abstract

Cloud computing has shown considerable growth due to its scalability and convenience in the past few years. With this change, a new programming paradigm called cloud-native originated. Cloud-native applications are often developed as a set of stand-alone microservices yet could depend on each other to provide a unified experience. Although microservices introduce many benefits regarding flexibility and scalability, it could be a great affliction to operate in production. Specifically, when operating a large system with hundreds of microservices interacting with each other, even the slightest problem could result in failures throughout the system.

The foci of this project are twofold. First, the authors introduce a robust Kubernetes native toolkit that helps researchers and developers collect and process service telemetry data with zero user instrumentation. Secondly, the authors proposed a novel way to detect anomalies by encoding raw metric data into an image-like structure and using a convolutional autoencoder to acquire the knowledge of the general data distribution for each service and detect outliers. Finally, a directed graph was used along with anomaly scores calculated before to show the spread of an anomaly to the user visually.

After an extensive testing and evaluation process, it was found that the telemetry extraction components are both resilient and lightweight even under sustained load. At the same time, the anomaly prediction algorithm is accurate and generalizable.

Keywords: AIOps, Monitoring, Disaster Recovery, eBPF, Kubernetes

Subject Descriptors: • Computing methodologies → Machine learning → Learning paradigms → Unsupervised learning → Anomaly detection • Computer systems organization → Architectures → Distributed architectures → Cloud computing

Acknowledgement

The completion of this research was an arduous journey, but it was also a fantastic learning opportunity. I am very lucky to be surrounded by inspiring people throughout my journey. I am grateful to a number of people who made my journey a special one. Without the advice and support of those who were there to reach out and help me to shine, I would not have been able to accomplish this research.

First of all, Mr. Guhanathan Poravi, my project supervisor, deserves a very special appreciation for helping me grasp what computer science research entailed, Kudos to him for guiding and instructing me through my path of success. It was his guidance and support that built my confidence and pushed me to ignite my highest potential with passion in learning. I would also like to express my gratitude to all the lecturers at the Informatics Institute of Technology for providing all the knowledge without any hesitation from the beginning of the degree programme. In extend to that, a very special thanks goes out to Jacob Payne, my Google Summer of Code mentor. He was the one who guided me through the ways of Kubernetes development and Matthias Rampke, who helped me in identifying the problem domain and building the scope of the project.

A special thanks goes out to all of the industry and academic experts who sacrificed their time to listen to my research presentations and provide feedback during requirement gathering and evaluation interviews for this research project. And finally, I have to thank my friends for encouraging me and motivating me during my difficult times and also to the seniors for their valuable feedback and assistance throughout the project. At Last but not least, I would like to thank my beloved parents for being patient and encouraging during all my ups and downs; Without you none of this would have been possible.

Contents

List of Figures

List of Tables

List of Acronyms

SRE Site Reliability Engineer

SLI Service Level Indicator

SRE Site Reliability Engineering

APM Application Performance Monitoring

MTTR Mean Time To Recovery

eBPF Extended Berkeley Packet Filter

VM Virtual Machine

AIOps Artificial Intelligence for IT operations

Gazer Telemetry extraction agent

Sherlock AI-engine

Operator Lazy Koala Resource Manager

1 Introduction

1.1 Chapter Overview

The introduction chapter offers an overview of the entire project. First, the author explains the problem domain of this research project, then proceeds on to the specific issue this project is going to address, then the motivation behind this project and its objectives, and finally concludes the chapter with the novelty of the research and expected research challenges.

1.2 Problem Domain

1.2.1 Cloud Computing

With the emergence of Infrastructure as a Service (IaaS), such as Amazon Web Services (AWS) and Google Cloud Platform (GCP), there is a big surge in organisations trying to outsource their computing needs to third parties (Rimol, 2021). This is mainly due to the elasticity given by all cloud providers. Users can easily scale up and down their infrastructure in minutes without making any commitments. All major cloud providers bill customers on the ”what you use is what you pay” model. Since the cloud provider manages the entire underlying infrastructure, customers do not have to worry about problems such as hardware failures. In contrast in a self-hosted setting if the user wanted one extra GB of memory than what’s available it requires a lot of effort and it costs a lot to fulfill that requirement.

1.2.2 Cloud-Native Applications

During the 90s and early 2000s, all the applications were made as a big monolith from a single code base (Spoonhower, 2018). Most of them were shipped as single binary. Since those days the applications were fairly simple, this worked divinely with little to no downsides. When the 2010s came around, there were many specialised frameworks and programming languages, and marketing teams wanted many new features quickly developed, still maintaining reliability (Di Francesco et al., 2018; Mike Loukides, 2020). Even then, if the code base of the application is stored in a single repository, developers have to go through a long process to review and test whether the changes will not break the current systems. Developers are also limited by the frameworks and programming languages that were initially chosen for the project.

To tackle these problems a new way to develop applications was introduced, it’s called ”Microservices”. The idea behind this concept is to break all functionalities of large monolithic applications into small individually scalable services and to give ownership of each service to a

clutch of people who work separately. With this flow, developers are free to use whatever tool they like to develop each service. Because these services are developed in parallel by different teams, this increases the development speed by an order of magnitude (RedHat, n.d.).

As these services are relatively small and tailor-made to run on cloud environments it's easier to take something that's running on the developer's local machine to the production cluster in a matter of minutes. This is mainly due to modern native cloud tools, such as CI / CD pipelines, which automatically build and test the code for them, which can save a lot of time by simply performing repetitive tasks that are prone to human errors (JetBrains, n.d.).

1.2.3 Monitoring Cloud-Native Applications

Even though cloud-native applications have a lot to offer when it comes to developer velocity and productivity, It has their fair share of issues. Most of these problems are linked to the sheer complexity of these systems and not having a proper way to monitor them (Zhitnitsky, 2019). All three major cloud providers provide a way to monitor these applications efficiently and some great open-source projects do this well, But to take full advantage of those systems, developers have to adapt their services to export all the vitals in a way the monitoring system understands. This works for most parts and is what all large companies are implementing; even if it takes more developer time, it is ultimately significant in regard to disaster recovery.

But there is still a slight problem with this approach. Once the system starts to scale up to hundreds of services, the number of vitals that has to be monitored goes to thousands and will require a lot of additional Site Reliability Engineers (SREs) and will have drop to lot of non-crucial service vitals and derive abstract Service Level Indicators (SLIs) to make it **humanly** possible to understand the current state of the system.

1.3 Problem Definition

One of the leading problems in monitoring microservices is the sheer number of data that they generate. It's humanly impossible to monitor the metrics of all the services and it's hard for a single person to understand the entire system. BySREs using abstracted metrics called SLIs which measures the quality of the service at a higher level can overcome this. Although SLIs can alert when there is an issue in the system, it is difficult to understand where the actual problem is from this point on. To understand the root cause of the problem, SREs need to dig into Application Performance Monitorings (APMs) of all the services and go through each and every

log of the troubling services.

When the system consists of hundreds or thousands of services that are interdependent; It is really hard to find where the actual issue is coming from and it may require the attention from all the service owners of the failing services to go through the logs and APMs to identify the actual root cause of the failure. This could greatly increase the Mean Time To Recovery (MTTR) and waste a lot of developer time by browsing logs.

1.3.1 Problem Statement

Modern distributed systems are becoming large and complex to the point where, when a failure occurs, it requires collaboration with a large number of people to find the actual root cause. Implementing a **framework** to make it easier to integrate machine learning models to detect anomalies in real-time could deflate MTTR.

1.4 Existing Work

1.4.1 Anomaly detection

Citation	Technology summary	Improvements	Limitations
Du et al. (2018)	Tested most of common machine learning methods to detect anomalies and benchmarked them	• Used SLIs to monitored data • A lot of good metrics (input data) • Performance monitoring of services and containers	• Only be able to identify predetermined issues • Require a sidecar that includes a lot of overheads • Won't work with event-driven architectures (this is where most of the new systems are headed) • Uses Supervised learning and it is nearly impossible to find real-world data with labels

Kumarage et al. (2018)	The authors here propose a semi-supervised technique using a Variational Autoencoder to predict future time steps and calculate the difference between predicted and actual to detect anomalies.	• Due to the difficulty of finding labelled research data, they settled on using a semi-supervised technique. • Randomized decision trees used were used to select the most suitable characteristics for each component.	• The model will not be effortlessly transformable for other systems • If more new key features were added to the system it will require a total retraining
Kumarage et al. (2019)	Uses a bidirectional GAN to predict future timesteps and uses MSE between prediction and real to determine the anomalies	Experimented using a GAN to detect anomalies rather than using conventional autoencoders	• Accuracy is around 60%, which is unimpressive for use in production with mission-critical systems. • As this is a GAN-based system, it may take numerous resources to run with production systems.

Table 1.1: Comparison of anomaly detection methods in distributed systems (self-composed)

1.4.2 Root cause identification

Citation	**Technology summary**	**Improvements**	**Limitations**
Gonzalez et al. (2017)	Detect failures in networks, using machine learning to generate knowledge graphs on historical data	• Predictable system • Automatic identification of dependencies between system events • Generalized to various systems	• Limited to network issues • SREs must manually identify the issues, although the knowledge graph helped visualisation.

Chigurupati and Lassar (2017)	Proposed a way to detect hardware failures in servers using a probabilistic graphical model that concisely describes the relationship between many random variables and their conditional independence	• Find hidden meaning in values that seems random • Used a probabilistic approach to understand the relationship between inputs and outputs more clearly • Gives all the possible root causes for a given problem	• Limited to hardware issues • Require support from domain experts • Can't account for unforeseen errors
Wu et al. (2020)	Find Performance bottlenecks in distributed systems using an attribute graph to find anomaly propagation across services and machines	• Created a custom Fault Injection module • Uses an attribute graph to localise to faulty service • Application-agnostic by using a service mesh • Rely on the service mesh to determine network topology • Uses unsupervised learning	• Only able to identify 3 types of issues • Checks only for performance anomalies • Use the slow response time of a microservice as the definition of an anomaly • Service meshes add a lot of overhead to systems • Required direct connection between services
Samir and Pahl (2019)	This detects and locates the anomalous behavior of microservices based on the observed response time using a HHMM	• Custom HHMM model • Self-healing mechanism • Focus on performance detection and identification at the container, node, and service level	• The input data set scale is limited • Require a sidecar • Needs to predetermined thresholds

Table 1.2: Comparison of root cause identification methods in distributed systems (self-composed)

1.4.3 Commercial products

Name	Futures	Limitations
Applied Intelligence by New Relic	• Metric forecasting. • Anomaly detection. • Alert grouping to reduce noise.	• Lack of explanation for certain classifications. • All the telemetry data need to be sent to a third party.
Watchdog by Datadog	• Monitor the metric data of the entire system from the background. • Monitor the log data. • Highlight relevant components affected by an issue.	• Announced in 2018 but is still in private beta. • Require code changes and tight integration with the datadog platform. • Available demos about the system seem to be designed for demonstration purposes.

Table 1.3: Comparison of commercial products for root cause analysis (self-composed)

1.5 The Novelty of the Research

1.5.1 Problem Novelty

After a literature survey, the author concluded that finding the root cause of any failure within a distributed system is a challenging issue. This is mainly due to the fact that this problem cannot be assigned to a fixed set of inputs and outputs, which is a basic requirement for almost all types of neural network that are readily available.

Most of the currently established research was done towards creating statistical models like clustering and linear regression. Although these algorithms perform outstandingly in small-scale systems, they can struggle to adapt in the large-scale, noisy monitoring data found in medium-to large-sized systems. Another problem that was recognised was that none of these articles adequately addressed the issue of constant changes in services. Most published research considers target services static, but, in fact, these services can change constantly within a day (Novak, 2016). Finally, non of the published work has addressed the issue of scaling the anomaly detection system relative to the target distributed system which is a crucial factor for production use.

1.5.2 Solution Novelty

The focus of this project is to create an adaptable and scalable series of components that ranges from instrumentation to root cause analysis, which can be well integrated into an existing system or can be extended to fit to newer use cases. To achieve this the author is utilizing a fairly new technique called Extended Berkeley Packet Filter (eBPF) for instrumentation, which is a Linux kernel API that can be used to track kernel events such as TCP socket changes to identify and understand the network layer of each application running on the system. Finally, for anomaly detection, a convolutional autoencoder with a novel data encoding method was used to keep the system as lightweight as possible, while still having acceptable accuracies for classifications. Combining that with a directed graph generated from collected network activity data can be used to highlight to blast radius of an anomaly along with possible causes.

1.6 Research Question

RQ1: How can a machine learning model improve MTTR in a distributed system?

RQ2: What is the most efficient way to present raw data monitoring to machine learning model?

RQ3: How to make an anomaly detection system scale relative to a distributed system of any size?

RQ4: What will be the most ideal machine learning model to uncover anomalies in a microservice?

1.7 Research Motivation

Modern distributed systems generate tons of useful and useless telemetry data. As the demand and size of the system increases, these telemetry data become more nosier and more complex (Khononov, 2020). It makes difficult for humans to understand all these data, especially if they do not have long-standing experience with the system. On the other hand, deep learning models thrive when they have no end of data to learn from. As these models can be trained in computer-simulated environments, they can learn concepts that humans need years to grasp, within days (OpenAI, 2018; Silver et al., 2017). Finally, unlike humans a deep learning model can monitor a service 24x7 without taking any breaks which will not only prevent outages even before they happen, It could be reduced MTTR because the issue can be detected way earlier than any human could do.

1.8 Research Challenge & Potential

1.8.1 Research Challenge

- **Highly seasonal and noisy patterns** - Monitoring metrics on microservices in production tend to have very unpredictable patterns depending on the traffic that has been sent to the service. The amount of traffic sent will depend on several external factors that are hard to determine. Modeling both temporal dependencies and static interdependencies found in telemetry data of services into a single graph will be very difficult and require a lot of fine-tuning and data engineering skills.

- **Overhead** - Modern deep learning models can solve any problem if we could give them an unlimited amount of data and processing power. Although in this case, the models need to optimize for efficiency over accuracy since having a monitoring system that consumes a lot more resources than the actual target system isn't effective.

- **Fit into Kubernetes eco-system** - Kubernetes has become the de facto standard for managing distributed systems (IBM, 2021). So the author is planning to create a Kubernetes extension that will bridge the connection between monitored service and monitoring model. But Kubernetes itself has a very steep learning curve, even the original developers themselves have admitted that Kubernetes is too hard and complex for beginners (Anderson, 2021).

- **Extraction of Telemetry** - Even though it's considered the best practice to implement telemetry exporting methods in the development phase of any application, developers often skip this part to save time. Sometimes it's required to depend on external applications that are developed by third parties which don't have means of exporting telemetry. Additionally, when building an end-to-end root course indication platform, it is required to take these kinds of scenario into account.

1.8.2 Research Potential

The feedback received for this project has been very positive because this is a very common but still unsolved issue in reliability engineering. Since this project was developed as a set of loosely coupled components, some of the experts expressed their interest in using individual components to solve some of the other problems they have been experiencing over time. Finally, this project can be used as an initial point for future research that focusses on specific areas of Artificial Intelligence for IT operations (AIOps) by replacing the individual components of this with their

own.

1.9 Research Aim

The aim of this research is to design, develop and evaluate a low overhead Kubernetes framework to collect, store and process telemetry data using deep learning to help system operators detect anomalies earlier in order to reduce the MTTR when the system is experiencing an anomaly.

In this project, the author tries to create a single model that can monitor all the vitals of a given service and output an anomaly score in any given time window. The author hopes to make it general enough so that operators can take the same model and deploy it with other services, and the model will be adopted to the new services using a few-shot learning method (Wang, Yao, Kwok and Ni, 2020). To make it happen, the author is trying to create a data encoding technique to represent monitoring data in a programming language/framework-independent way. To achieve this goal the author is also hoping to create a lightweight service instrumentation pipeline that can collect and process telemetry data in real-time without requiring any additional work from the user's end.

1.10 Research Objectives

Research Objectives	**Explanation**	**Learning Outcomes**	**Research Questions**
Problem identification	When selecting the problem the author wanted to pursue, they had three main goals. RO1: The problem domain should be something they would enjoy working on. RO2: At the end of the research they should give a meaningful impact to the target domain, both in the theoretical and practical aspect. RO3: It should be challenging to achieve, and the results should speak for themselves.	LO1	RQ1, RQ3

Literature review	Conduct a Literature review on root cause analysis, RO4: To find the current methods that are used for anomaly detection and root cause localization. RO5: Uncover issues with current approaches. RO6: Understand how advancements in other related domains can apply to this domain.	LO3, LO4, LO6	RQ1, RQ2, RQ3, RQ4
Developing an evaluation framework	During the literature survey, one of the problems the author identified was that there is no uniform data set when it comes to training or evaluating models to detect anomalies in microservices. Most of the researchers used private datasets to train and test their work. To address this, the author is developing RO6: A tool that can easily simulate a distributed system in a cloud-native setting. RO7: A tool that can inject anomalies into the running services.	LO7	RQ4
Data gathering and analysis	To create a model to detect anomalies, the author will have to RO8: Simulate a distributed system. RO9: Simulate the traffic inside the system. RO10: Collect monitoring data while it's running.	LO7	RQ2, RQ4
Developing the encoding method	Since these microservices will report very different metric values even at idle depending on the architecture of the service. To normalize theses data points from all the services to one format author will, RO11: Evaluate current data encoding methods such as Zhang et al. (2019). RO12: Find the best fit and optimise it for this use case. RO13: Test if there is improvement by using that method.	LO2, LO5, LO7	RQ2, RQ3

Developing the model	According to Kumarage et al. (2019), autoencoders tend to perform best when it comes to anomaly detection. But during the literature survey it was revealed that convolution autoencoders weren't tested for this usecase. So, the author is hoping to develop a convolution auto-encoder and test how it will perform.	LO2, LO5, LO7	RQ4
Testing and evaluation	The following things will be tested during the testing phase, RO14: How will the system classify long-term & short-term fluctuations. RO15: What will be the overhead of the system. RO16: Can the system understand the mapping between core metrics like CPU and Memory usages. RO17: Accuracy of fault detection. RO18: Reliability of the instrumentation system.	LO8, LO9	RQ1, RQ3, RQ4
Integration	Having a fancy model does not add anything if it is very hard to use with a real system. So the author is hoping to develop a Kubernetes extension that will map the model with any service given by the user.	LO7	RQ1, RQ3

Table 1.4: Research Objectives (self-composed)

1.11 Research Contribution

1.11.1 Domain Contribution

With this research, the author first tried to develop a **cloud-native solution to create a configurable microservices system**, so this research and future research will have a standard environment to develop and evaluate their work. The author built a lightweight and **low-overhead data collection pipeline** using eBPF to collect telemetry of target services without any instrumentation from the user.

1.11.2 Knowledge Contribution

One of the main problems with monitoring microservices systems is that different services can be developed with different programming languages and frameworks, and those can contain different levels of noise. Therefore, it is complicated for a single model to detect anomalies in any service since some frameworks tend to use more resources while idle than others. To address this author is trying to come up with an **encoding method** so the model can be trained to monitor one framework and those learning will still be valid for another framework. With these encoded data, the author is hoping to develop a **convolutional autoencoder that will use unsupervised learning to spot out anomalies in a given data stream**. This may have better performance while using fewer resources since convolutional layers are typically lightweight and good at pattern recognition (Oord et al., 2016). Finally, the author plans to aggregate these predictions from the models into a pre-generated service graph and weigh them at **find all possible root causes**.

1.12 Project Scope

As stated in the literature survey and after discussing with industry experts, the author found many issues that could be addressed when developing this system, but some of those problems, like getting the models well optimised to react to the anomalies, is a complex task to achieve. Therefore, the main focus of this project is to create an end-to-end framework which will make the application of machine learning to solve Site Reliability Engineering (SRE) issues straightforward, so future work can be built on this project.

1.12.1 In-scope

Following are the main focusses of this project.

- Low overhead data collection pipeline to collect service telemetry.
- Loosely coupled architectures; so that all components can be individually deployed.
- UI to easily visualise the service topology and identify root causes.
- Optimized models that have a small memory footprint and CPU overhead.
- Well-generalised model which will be able to be deployed with completely new services and will learn to adapt to the new system.
- Scalable to fit with a systems of any size.
- Easy adaptability.

1.12.2 Out-scope

Following will not be covered during this project

- Working outside of Kubernetes eco-system.
- Monitor websocket connections.
- Acts as a fully functional monitoring system.
- Works with Multi-Cluster setups.
- Highly accurate predictions from the model.

1.12.3 Prototype Feature Diagram

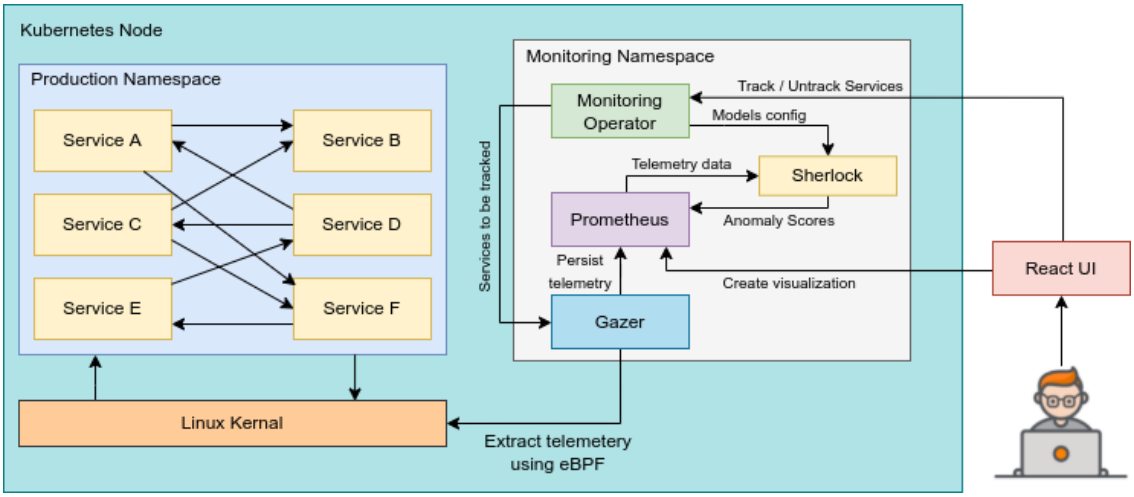


Figure 1.1: Prototype feature diagram (self composed)

1.13 Chapter Summary

This chapter gave an overview of cloud computing and how it matured overtime along with one of the most pressing issues that currently holds all the cloud native applications. Then the author described his plans to tackle this problem and listed how it can be achieved along with the unique challenges he had to overcome in order to complete this project.

2 Literature Review

2.1 Chapter Overview

Since most research projects are continuation or different approaches to existing problems, one of the first most crucial steps in conducting research is to conduct a literature survey. This usually has to be done to understand the problem domain, currently published work about the problem, established methods of solving target and related problems, tools and technologies that can be used, and finally evaluation methodologies for that particular problem. So in this chapter, the author will discuss how this problem came to be, what are the different components that needed to be developed and why, and finally conclude the chapter with possible approaches and tools that can be used to solve the aforementioned problem.

2.2 Concept Map

During the literature survey, the author has analysed more than fifteen different research publications and articles to get a deeper understanding about the problem domain. The visualization can be found on Appendix A.

2.3 Domain Overview

2.3.1 Introduction to Distributed Systems

Distributed systems are a type of system that is designed to operate in a fragmented setting. This fragmented style helps the system to distribute its workload over many computers across a network, which in itself makes scaling such a system easy, as adding more computers to the network. This method of scaling is called horizontal scaling. Using this kind of fragmented architecture helps to increase the reliability of the system, since the likelihood of a single hardware failure knocking out the entire system gets smaller and smaller as the network grows.

In the early days, only large-scale enterprises could afford the cost of building distributed systems, but in recent years with the rise of cloud computing (Galov, 2021), creating our own distributed systems could be done with just a few clicks of the buttons.

2.3.1.1 Microservices

This radical shift introduced a new paradigm of computing called microservices, where a bunch of small self-containing services work together to form an enormous and complex system. These services can be individually deployed and scaled. Due to this nature users can deploy replicas of

a single service across many Virtual Machines (VMs) and put a load balancer that will split the traffic between them. This method will allow the service to maintain availability even if multiple VMs which contain a copy of the service goes down (Chaczko et al., 2011).

2.3.1.2 Containerization

Although microservices helped more organisations to meet a higher level of availability due to their decoupled nature; It was perplexing to manage the loads of diminutive services to spread across hundreds of VMs. Since these services were isolated using a virtualization layer, the cost and the overhead of maintaining a system as mentioned were very high (Dua et al., 2014). To mitigate this problem inspired by the logistics industry, a new method to package an application called ”Containerization” was invented. The rationale behind this technique was to package all dependencies of the program into a single image without the operating system itself and, when running, to share a single logically separated operating system across all the containers. Therefore, using this technique will reduce a lot of overhead in the system and also make it simple to move an application running on a local computer to a remote server due to the dependency packaging technique.

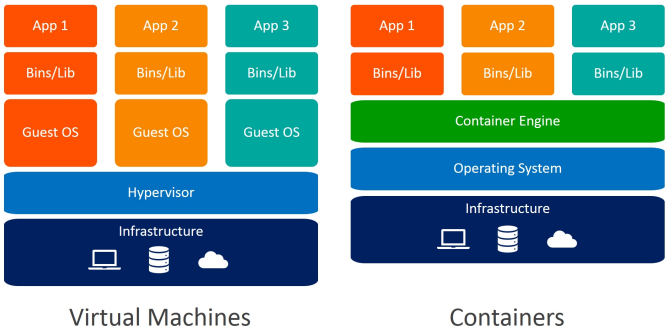


Figure 2.1: Difference between hosting 3 apps in virtual machines vs Containers (WeaveWorks, 2020)

2.3.1.3 Container Orchestration

Although containers solve major dilemmas when it comes to operating a distributed system, managing 100s of containers becomes a major challenge. The reason for this is when it comes to a large-scale distributed system, it’s simply impossible to use one VM to house all the containers. These containers should be spread over doses of VMs and in some cases different VM vendors. Then there are networking, replication, security, and **monitoring** to consider. To solve all of these problems number of different container orchestration systems were introduced (al dhuraibi et al., 2017). But due to a survey done by Red Hat in July 2021, it is revealed that 88% of

users prefer to use Kubernetes as their container orchestrator while 74% mentioned they use Kubernetes for their production workloads (RedHat, 2021).

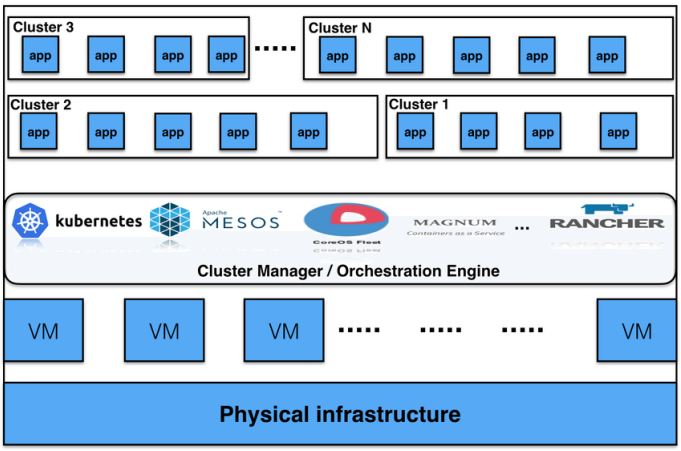


Figure 2.2: Overview of a container orchestration engine (al dhuraibi et al., 2017)

2.3.2 Reliability Engineering

With the rise of cloud computing, a new culture of software development called DevOps emerged. According to Kim et al. (2014) the philosophy behind DevOps was adopted from the ”Toyota Way” (Liker and Meier, 2006). In that book, the author talks about ”The Three Ways”;

1. **Principles of Flow** - Workflow from left to right (from requirement to production) and to maximise flow batch sizes must be reduced.

2. **Principles of Feedback** - To increase quality, feedback must be passed from right to left so that the whole idea to the production workflow has a feedback loop.

3. **Principles of Continuous Learning** - Every failure is a learning opportunity.

These three principles essentially constitute the modern DevOps culture. Therefore, a well-implemented DevOps culture on an organization will yield much better results in both the quality of the system and its reliability.

As mentioned above DevOps is considered as a culture or set of abstract principles that breakdown the organizational silos to achieve a higher level of agility that was considered as impossible some time ago. SRE, is the implementation of this abstract concept with clearly defined roles and tasks. Their main responsibility is to keep the system running smoothly as much as possible and to adapt the infrastructure to fit the needs of the system. To achieve this SREs rely on number of automation tools, which help them from building the software to extracting real-time insights while it is running in production.

2.3.3 How to Identify the Root Cause of a Failure

To identify the root cause of an issue in the system, three major steps are needed to be passed.

1. Detect whether there is an issue with the system.
2. Find all the affected services.
3. Estimate the most probable cause.

Typically, most of the distributed systems have some sort of a monitoring system which collects telemetry data about the system in real-time. This allows the SREs to get a bird's eye view of the system's status. But to achieve a higher level of reliability it is crucial to keep tabs on every sub-component of the system, So it's possible to get an understanding on its behavior at anytime.

Even though this is the ideal scenario, this approach doesn't scale well. At some point, it's getting humanly impossible for the SRE team to keep track of all these services. So, to solve this concept, SLI was introduced (Beyer et al., 2016). The idea behind this provides a quantitative measure of a very specific part of the system as faced by end users. For example, request latency is one of the most widely used SLI. This helps to lower the number of metrics SREs has to monitor but minor errors that affect a minority of users could go unnoticed for months.

To detect these kinds of issues, all the microservices in the system need to emit a lot of telemetry data and those data need to be individually processed in near real-time to catch errors early. The most widely adopted method of extra meaningful data from such data stream is setting up threshold-based alerts which will send notifications when there is a threshold violation. The main drawback to this issue is that SREs has to predict both metrics that need to be monitored and "normal rates" for these metrics by looking at past data and that value has been valid for the future as well, if the service is newly deployed, it is really hard to get those two right. In fact, on 14 December 2020, all of Google's services went unresponsive due to Google's OAuth service running out of disk space, and no one at Google did not notice until user reports started flooding in (*Google Cloud Incident #20013*, 2020).

Typically, in distributed systems, services have many interdependent connections to form a larger system. When a dependant service is experiencing an issue for example the elevated level of request latency, it is possible for a service consuming that service to also show an elevated level of request latency. So when there is an outage or system issue SREs, look at all services

with abnormal metric readings and try to make an educated guess of the most probable root causes. This is repeated until an accurate root cause is found. The hardest part of this process is making educated guesses of the most probable root causes, and this requires the participation of a system expert with a lot of experience with the target system.

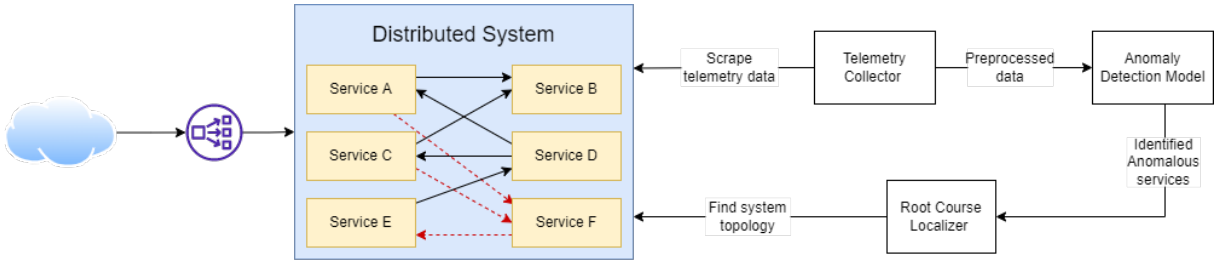


Figure 2.3: Root cause localization pipeline (self-compose)

2.3.4 Artificial Intelligence for IT operations

When it comes to IT operations, such as managing infrastructure and identifying the root cause of failures, as mentioned in Section 2.3.3, there tend to be many structured and unstructured data sources. AIOps is an emerging field (Linders, 2021) where both data scientists and reliability engineers cooperate to build smarter, data-driven systems with the help of machine learning to achieve a higher quality of service, which is not simply possible by using traditional methods due to the density and complexity of the data sets.

2.4 Technologies

2.4.1 Monitoring Techniques

2.4.1.1 Process Monitor

One of the most basic forms of monitoring applications is called system monitoring (Wiesen, n.d.), where the target application is monitored using the operating system's process manager which keeps track of resource usages from each of the running processes. This works well for small monolithic applications where there is only one application per server.

2.4.1.2 Hypervisor Based Monitor

Around the time of the early 2000s, a new way of managing servers called virtualisation became popular (*What is virtualization?*, n.d.). The idea behind this was to have one big server split into smaller isolated VM which made sharing and managing hardware resources easier. To achieve this, the host computer must run a software called Hypervisor which can create and delete guest VMs on demand (Mergen et al., 2006). As the hypervisor is responsible for managing the VM's

resource requests, it can be used to observe the resource usage. Most cloud providers still rely on hypervisors to create VMs for customers (Andy Honig, 2017). One of the most commonly used hypervisor is called Kernel-based Virtual Machine (KVM) which uses the Linux kernel itself as the hypervisor, Which has a very small overhead on the host machine and grants better visibility into the guest system (Kivity et al., 2007).

2.4.1.3 Service Mesh

When the containerised distributed systems started to become popular, both developers and operators were required to gather more insights about each application. Manually programming each application to collect telemetry is a time-consuming task, and developers were reluctant to implement these. Service Meshes were introduced to overcome this challenge (Li et al., 2019). The idea behind this is to keep target service as it is and build a wrapper around it which will do all the instrumentation for the service. This is usually implemented as a side-car proxy (Gillis, 2019).

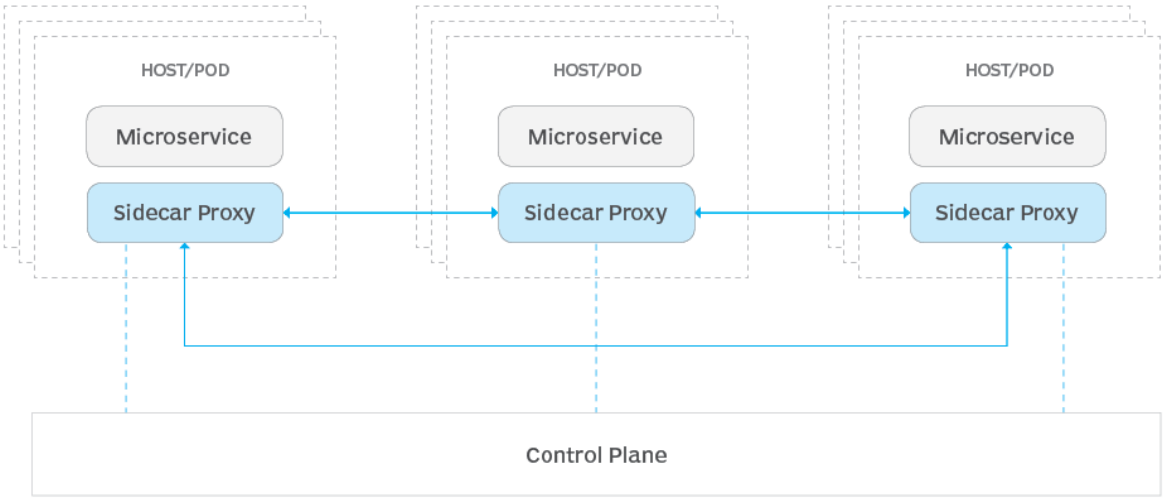


Figure 2.4: Sidecar proxy design pattern (Gillis, 2019)

As this proxy sitting outside of the service, it is language-agnostic and intercepts all the inbound and outbound traffic at the application layer and relays them to the responsible parties. This is a very effective method to provide visibility into services without requiring additional work from developers. The main problem with this approach is that it adds a lot of network and CPU overhead to analyse requests with proxy (Morgan, 2021).

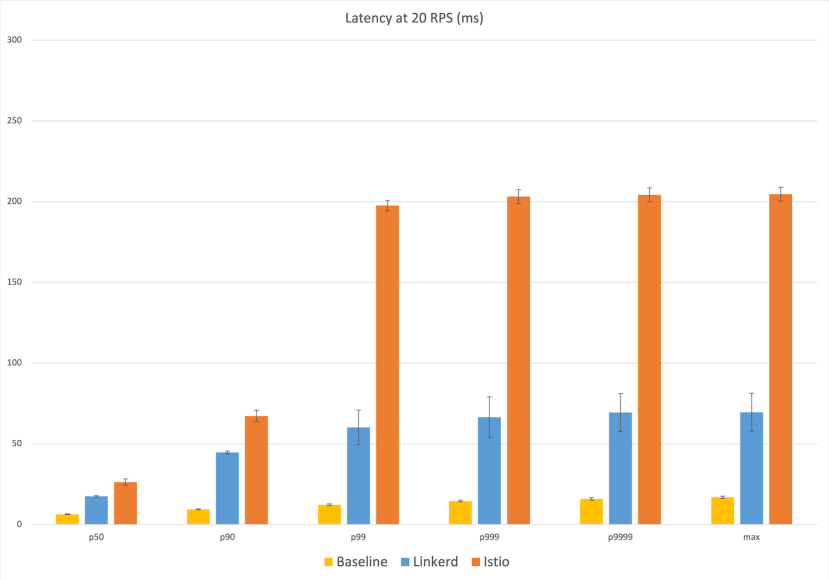


Figure 2.5: Service mesh benchmark (Morgan, 2021)

2.4.1.4 Extended Berkeley Packet Filter (eBPF)

eBPF is a feature introduced to the Linux kernel in version 3.15 that allows deploying sandboxed programs to kernel space at runtime which could be used for application instrumentation from the kernel level (Molnar, 2015). eBPF essentially created a way to run hooks on kernel events. For example kernel method ”tcp_v4_syn_recv_sock” gets called every time a client wants to establish a TCP connection with the server. With eBPF, users can deploy a lightweight hook that is called every time a new connection is made which will update the state on eBPF map that could be read from user-space.

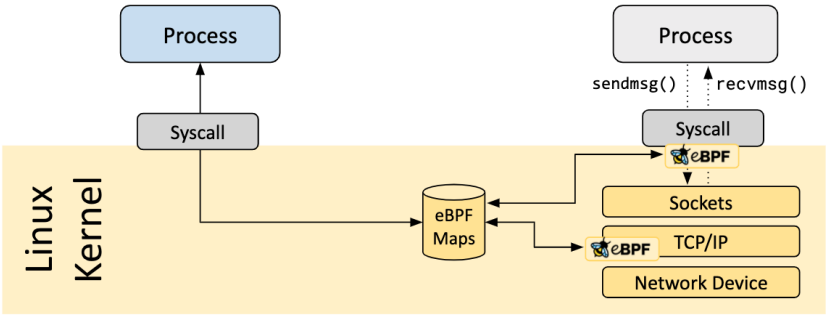


*Figure 2.6: eBPF architecture (*What is eBPF?*, n.d.)*

Using these data, the request rate of a given service can be calculated with minimum overhead and zero instrumentation on the application. The main drawback of this method is it is complicated to capture high-level data like HTTP status codes since those data only exist at the application level as it is.

Technique	Advantage	Disadvantage	Citations
Process Monitor	Virtually no overhead. Works out of the box.	A very limited number of data points. Not suited for a distributed system.	Chigurupati and Lassar (2017) Kumarage et al. (2018) Kumarage et al. (2019)
Hypervisor Based Monitor	Most cloud providers provide a simple API to access these data.	A limited number of data points	Du et al. (2018) Geethika et al. (2019)
Service Mesh	In-depth monitoring, Request analysis, and modification at fly Framework Independent	Performance Overhead	Samir and Pahl (2019) Wu et al. (2020)
Extended Berkeley Packet Filter	Very low overhead Works at the kernel level Able to scrape data related to the system	Works only on Linux-based systems Difficult to develop and use	None

Table 2.1: Comparison of instrumentation methods (self-composed)

2.4.2 Detecting Anomalies

2.4.2.1 Supervised Learning

In general, The most popular way to detect anomalies is using supervised learning methods. Supervised learning methods can be implemented from finding outliers in sales patterns to fraud detection. Du et al. (2018) As mentioned in the cloud computing domain, more than half of the methods are still based on supervised learning to detect anomalies. Among these Support Vector Machines (SVMs), Random Forest and Decision Trees were used the most. But one of the main downsides to using supervised learning in cloud computing is the lack of labeled anomalous data. Since most of the systems nowadays target at least 99% of uptime finding labeled data is difficult. Even if there is a well-balanced dataset, the trained model won't be able to recognize unforeseen anomalies.

2.4.2.2 Semi-Supervised Learning

As mentioned in 2.4.2.1, one of the most challenging issues with anomaly detection is to find a data set with enough labelled abnormalities. One of the key factors contributing to the development of a well-generalised model is having a well-balanced data set (Batista et al., 2004). If the authors managed to find a sizable unbalanced data set, using clustering algorithms like K-nearest Neighbours (KNN) could yield better results. Akcay et al. (2018) managed to utilise an encoder-decoder-encoder architectural model to detect anomalies in image data and achieved remarkable results.

2.4.2.3 Unsupervised Learning

When the target data set consists of a lot of unlabelled data and is difficult to label by hand, machine learning experts lean towards using unsupervised learning so the model can be its own teacher (Mishra, 2017). Silver et al. (2016) managed to develop the first AI that was able to beat the best Go player in the world with a score of 4-1. This model learns to play the game of Go by looking at thousands of games played by humans and learning to approximate the optimal strategy to any given board position. One year later same authors released the updated version of the model which learn to play Go without any human interference and this model beated the previously published model 100-0 (Silver et al., 2017). This proves deep learning models could even surpass humans when it comes to finding patterns in very large distributions.

Technique	**Advantage**	**Disadvantage**	**Citations**
Supervised Learning	Easy to develop and train. Models will converge better to the dataset. Easy to test the model performance. Ideal used for classification and regression problems.	Require a labelled dataset. The models will not look at scenarios outside of the dataset. The model will be biased if the labelled dataset was biased.	Du et al. (2018)

Semi-Supervised Learning	A blend of both Supervised and Unsupervised learning. Developers can force the model to learn some behaviors.	Require a labeled dataset to train the initial steps. Model isn't free to understand the problem from the ground up.	Akcay et al. (2018)
Unsupervised Learning	Doesn't require a labeled dataset. Excel at clustering extracting patterns from datasets.	Developers have no control over the behaviour of the model.	Kumarage et al. (2018) Zhang et al. (2019) Kumarage et al. (2019) Khoshnevisan and Fan (2019)

Table 2.2: Comparison of anomaly detect methods in distributed systems (self-composed)

2.4.3 Root Cause Identification

It's very difficult to use a standard learning algorithms like Multilayer Perceptrons (MLP) to predict faulty service because the number of microservices in distributed systems changes frequently. Even if the system were to retrain the model after every new deployment it will hesitate to predict newly added services as the root cause since it does not have any historical data about the service to make assumptions. Therefore, almost all published research uses the Key Performance Indicator (KPI) correlation or some variations of graph-based methods to predict the root cause of failures (Soldani and Brogi, 2021).

Technique	**Advantage**	**Disadvantage**	**Citations**
KPI correlation	Could find indirectly affected services. Easy to implement.	The search space is large. Could result in a lot of false positives and noisy outputs.	Nguyen et al. (2011) Nguyen et al. (2013) Wang, Zhao, Chen, Li, Zhang and Sui (2020)

Graph-based methods	Gives clear visual reasoning for the predications.	Miss out on indirectly affected components. Computing the causality graph could be expensive.	Samir and Pahl (2019) Wu et al. (2020) Ma et al. (2020) Meng et al. (2020)

Table 2.3: Comparison of root cause identification techniques (self-composed)

2.4.4 Evaluation

Since this project consists of 3 components working together, finally all of these can be evaluated separately as the whole system.

To evaluate an instrumentation system, the author is hoping to use a static load generator like MicroSim to simulate a small microservices system and measure its performance of it by extracting CPU and Memory usage. Then gradually increase the complexity of the system and measure how the CPU and Memory usages change over the complexity of the system.

When it comes to data science, the data science component of this project is based on an unsupervised convolutional autoencoder. This model is trained on telemetry extracted from the system under normal behavior. The learning goal given in the model was to take the input and reconstruct it to the best of its ability. Due to the architecture autoencoders doing, it will force the model to learn the relationship between each metric and how it corresponds with each other. Since the model's output never be a perfect recreation of the input traditional metrics like Precision, Recall, and F1 Score can not be used. Instead, the only metric that can be tracked during the training and testing is the reconstruction loss (Ghasedi Dizaji et al., 2017). The final output presented to the user end of this research will be the servie's predicted health. This value will be derived by getting the difference between the highest and lowest possible value for reconstruction loss and calculating which percentile it belongs to.

2.5 Existing Systems

As the large-scale migration towards the cloud and microservices started recently, the problem this research is trying to solve mainly affects large-scale enterprises there are not a lot of published research on this domain. All the work done towards uncovering the root cause of failures by huge co-operations either kept their finds for internal use to sell them as Software as a service (SaaS) product.

One of the best implementations found on root cause analysis is from Datadog. They created a platform called watchdog (Menezes, 2018) that monitors the entire system for anomalies and failures in the background. When a failure occurs it tries to pull all the relevant stack traces and monitors data to a single view so the developer can diagnose the problem easily. The problem with this solution is that even though it was announced back in July 2018, all that is available is currently in private beta which not everyone has access in.
All the currently published work on microservices monitoring can be classified into 3 main categories

1. Instrumentation
2. Anomaly detection
3. Root cause identification

2.5.1 Instrumentation

One of the first steps that needs to follow to get visibility into running service is to set up a data collection pipeline that collects performance data about the service and writes to persistent storage for later evaluation.

Currently, the most popular way to collect telemetry data on microservices is using an open-source tool called Prometheus which has created by SoundCloud and later donated to Cloud Native Computing Foundation (CNCF). Usually, Prometheus has matched it Grafana visualized these data to make educated guesses. Toka et al. (2021) proposed Kubernetes native a data analytics pipeline that uses Prometheus for data scraping and runs a real-time analysis on them. The issue with this approach is to get some key data like the number of inbound and outbound requests; the service is required to be architectured to work with Prometheus. If it is third-party software without Prometheus integration, the system is limited to scraping metrics like CPU and Memory usage, which are exposed by the operating system.

2.5.2 Anomaly detection

The detection of anomalies in time series is a field of its own. According to Hagemann and Katsarou (2020) anomaly detection in cloud computing environments date back to 2012 which are mostly based on statistical approaches. Since 2014 their big shift toward using machine learning-based approaches to detect anomalies. Due to the sheer number of data points modern systems generate and the complexity of those data.

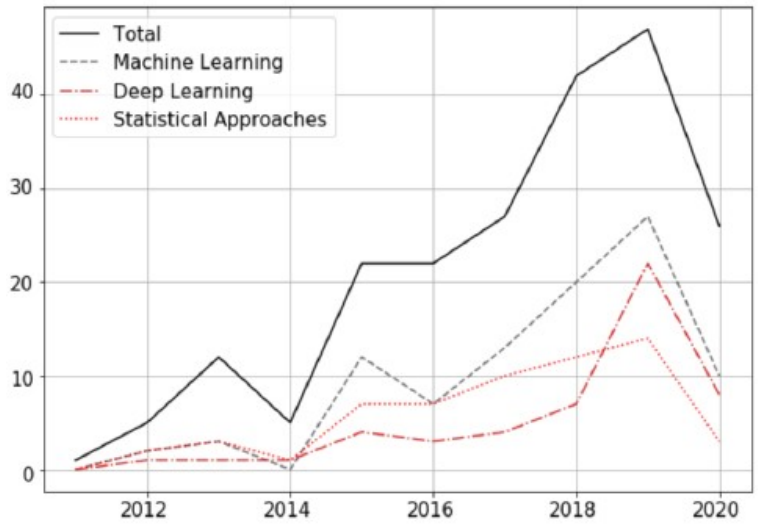


Figure 2.7: Number of papers published work on anomaly detection in cloud computing environments (Hagemann and Katsarou, 2020)

2.5.2.1 Standard machine learning

Du et al. (2018) tried experimenting with four of the most popular machine learning techniques to detect performance anomalies. To do this they used an open-source virtual IP Multimedia subsystem called Clearwater. Their system had three modules monitoring agent, data processing, and fault injector. To run the experiment, they first used a load generator to create traffic within the system. Then, they used the monitoring agent to collect telemetry data while the fault injector introduces random faults into the system. Finally, they combined the data from the monitoring agent and the fault injector to create the data set to train four machine learning models. After training, each model was a plugged-in to the data processing module and tested its precision, recall, and f1-score. In the end, they concluded that the K Nearest Neighbours classifier gives the most accurate classifications, while Support vector machines have the worse.

2.5.2.2 Encoder-decoder networks

Auto-encoder

Auto-encoders are a type of neural network that will try to predict the input data from itself. To do that, the model must learn how to pass through the input data to the output layer through a bottleneck layer. After training this bottleneck layer is called latent space which compressed low-level representation of the entire input data distribution (Hinton and Salakhutdinov, 2006). Therefore, the job of the autoencoder is to understand the underlying pattern in the given data distribution.

Due to this nature, autoencoders have become a popular method of detecting anomalies

in time-series data, since to find anomalies from input sequence one has to learn to identify the normal and the abnormal. After training the model on the target dataset, it should be able to come up with the generalised function for the given dataset and it will be able to recreate any input sequence accurately. However, when there is an anomaly in the input sequence models, the output will be vastly different from the input because the model doesn't know how to recreate it properly. This reconstruction loss can be used as a metric to uncover anomalies within the system. In Kumarage et al. (2018) authors used this method to detect anomalies in distributed systems. The main benefit of this approach dataset does not have to be labelled and the model learns to differentiate normal from the abnormal.

Generative Adversarial Networks (GAN)

In a continuation of their work Kumarage et al. (2019) they tried to do the same thing but on the opposite side using a GAN (Goodfellow et al., 2014). Here the generator network is trying to learn the targeted data distribution (non-anomalous dataset) while the discriminator tries to classify the normal and abnormal. In this, the generator network acts as the decoder, while the discriminator network tries to encode the generator's output into a single scalar value, which is the anomaly score. The authors of the paper tried using a traditional GAN and a bidirectional generative aversarial network (BiGAN) (Donahue et al., 2016), but in the end they concluded, although it showed a tendency towards better performance when the dataset becomes larger, with the dataset they had auto-encoders performing well overall.

2.5.2.3 Convolutional neural networks

Ever since DeepMind came up with wavenet which used a CNN to generate audio samples (Oord et al., 2016) researchers uncovered other potential use cases other than image-related tasks. One of those use cases was that CNN excels at pattern recognition, encoding time series data sets into image-like data structures, and using a CNN to identify abnormal patterns in it. On Kim et al. (2018) the authors tried to use a novel technique to encode raw data in a pixel-like structure and found that it could outperform existing methods to detect anomalies in computer networks. In another work by Zhang et al. (2019) Convolutional Long-Short-Term Memory (ConvLSTM) with an attention mechanism, which yielded more accurate temporal dependencies.

2.5.2.4 Root Cause Identification

Predicting the exact root cause of the failure using just a standard machine learning model is a pretty difficult task, since the prediction space is not finite. In 2017, a team from Google X tried

using the Bayesian Network to model the relationship between the state of the system and its effect on failures (Chigurupati and Lassar, 2017). Using it, they were able to accurately predict all the possible root causes of a hardware failure in certain systems but this model required to predefine all the possible error modes by domain experts, which is not possible in a constantly evolving distributed system. There were similar attempts Gonzalez et al. (2017) to use machine learning to generate knowledge graphs on historical data and help developers develop reasoning for failures, although this eliminated the need for a domain expert, and this cannot also react to unseen errors.

In a distributed system, it's hard to spot real anomalies by looking at monitoring data. But when there are huge spikes in response latencies or error rates, it is a good indicator that something is wrong. So Samir and Pahl (2019) used a Hierarchical Hidden Markov Model (HHMM) to uncover the possible affected services from changes in response time or error rates in one service and used those data to reveal the root cause of the issue. All the papers discussed above have one problem in common. They all assume the entire system is static, but the reality is that these services change over time either with increased demand or new future implementations. In addressing this, Wu et al. (2020) developed a service that monitors all operating applications and their vitals. This also constructs an attributed graph that represents how each service interacts with the other. When the monitoring system detects an anomaly MicroRCA weight that graph with response time changes and tries to find the epicenter of the anomaly. The problem with these approaches is the authors rely solely on slow response time as an indication of an anomaly but several other factors could course anomalous behaviors without changes in response times.

2.5.3 Comparison of Existing Systems

Research	Improvements	Citations
Instrumentation		
Toka et al. (2021)	• Explained a way to build a cloud-native data aggregation and analytics pipeline using open-source software. • The proposed system is not platform dependant.	• The overhead on the monitoring system is a bit high. • Analytics pipelines rely solely on KPI correlation.
Anomaly detection		

Prabodha et al. (2017)	• Explained the most popular methods to detect anomalies.	• The authors didn't consider learning-based approaches.
Kumarage et al. (2018)	• Due to the difficulty of finding labelled research data, they settled on using a semi-supervised technique. • The random decision trees used were used to select the most suitable features that correspond to each component.	• The model won't be easily transformable for other systems. • If more new key features are added to the system, it will require total retraining.
Kim et al. (2018)	• Introduced a new encoding technique so that CNN can identify anomalies better.	• Although this outperformed the 'grey scale encoding' technique (Dasgupta and Majumdar, 2002), a comparison study with random forests showed that this method is outperformed by random forests.
Du et al. (2018)	• Introduced SLIs to monitor data. • A lot of good metrics (input data). • Monitoring of the performance of services and containers. • Tested multiple Machine learning models to see which one works best. • Introduce a Fault Injection System.	• Only can identify predetermined issues. • You need a sidecar that includes a lot of overhead. • Won't work with event-driven architectures. • Uses Supervised learning and it is nearly impossible to find real-world data with labels.
Kumarage et al. (2019)	• Used a different approach to predict the future timesteps from the past events. • Which outperformed passed techniques when the monitored data points become larger.	• Accuracy is around 60% which is not good to use in production with mission-critical systems. • As this is a GAN-based system, it may take a lot of resources to run with production systems.

Zhang et al. (2019)	• Introduce a new embedding technique. • A hybrid method used that uses the convolutional autoencoder and LSTM network.	• Feature maps are not iterative to understand, so operators must blindly trust the network. • It will be hard to set the network to ignore expected anomalies.
Root cause identification		
Chigurupati and Lassar (2017)	• Experimented with different metrics till it narrowed down. • Find hidden meaning in values that seems random. • Using a probabilistic approach to better understand the relationship between inputs and outputs. • Gives all the possible root causes of a given problem.	• Require support from domain experts. • Can't account for unforeseen error.
Gonzalez et al. (2017)	• Build a predictable system. • Automatic identification of dependencies between system events. • Doesn't Need to rely on Domain experts. • Generalized to different systems. • Data windowing used. • Randomly permuting to test how the model reacts to random inputs.	• The knowledge graph is good for visualisation of the problem, but people still have to manually figure out what went wrong. • It assumes that the collaboration and knowledge of network operators and managers are available. • Random Forests does not scale well for a big input set.

Wu et al. (2020)	• Create a custom fault injection module. • Uses an attribute graph to localize to faulty service. • Application-agnostic by using a service mesh. • Rely on service mesh to determine network topology. • Unsupervised learning.	• Only able to identify 3 types of issues. • Looks only for performance anomalies. • Use the slow response time of a microservice as the definition of an anomaly. • Service meshes add a lot of overhead to systems. • Paper doesn't talk about services that ain't directly connected. • Won't work with event-driven architectures.
Samir and Pahl (2019)	• Custom HHMM model. • Self-healing mechanism. • Focus on the detection and identification of performance at the container, node, and microservice level.	• The input dataset scale is limited. • Require a sidecar. • Needs predetermined thresholds.

Table 2.4: Review of existing systems (self-composed)

2.6 Chapter Summary

In this chapter, the author discussed the origin and evaluation of SRE and AIOps and how that paved the way for the problem that's being tackled with this project. Then the author went on to discuss the techniques and tools that can be used to find answers to the problem of anomaly detection and the root cause analysis in distributed systems while listing down the pros and cons of every technique while citing sources for these claims. Finally, the author showcased the three main components of automated root cause analysis using the published literature and finally concluded the chapter with a comparison of the most relevant work for each of the identified components.

3 Methodology

3.1 Chapter Overview

This chapter focusses on the research, development, and management methodologies of this research project. It starts by describing every research methodology that was chosen and considering the reasoning behind them. Then move on to explain how the final product will be developed. Finally, this concludes with a justification for the project management methodologies.

3.2 Research Methodology

Research Philosophy	Mainly, there are four research philosophies, Pragmatism, positivism, realism, and interpretivism. It explains the belief, and research is done. After doing an in-depth study on research philosophies, the author decided to follow **Pragmatism** as research philosophy because the author believes there is no one way to solve the problem and this research is trying to address that the goal of this research is to solve a practical problem faced by SREs. (Munim (2019), Dudovskiy (n.d.))
Research Approach	Although the inspiration for the research came from an observation of the real world; The author is using **deductive reasoning** to approach the problem. After the problem was identified the author looked for existing work, found few theories on the domain. The author then found few flaws in these methods and thought of a way to address them with different approaches. At the end of the research other hopes to implement these new approaches and observe their outcome.
Research Strategy	The research strategy will be used to answer the research questions. In this project, the author will use **experimenting, interviews, and surveys** to provide answers to research questions.
Research Choice	During this research project, the author is planning to build a very generalized solution to predict anomalies. Therefore, to achieve this, a **quantitative** dataset will be used to train the model, while a **qualitative** data set will be used to evaluate it. So, the data for this research will be collected using **Mixed method**.

Time zone	This project needs to be completed within 8 months, so a **cross-sectional** time horizon will be used to collect data to complete the project.

Table 3.1: Research methodology selection (self-composed)

3.3 Development Methodology

Even though this project has a few clearly defined requirements, designing and developing them will require an iterative model as there is not a single best way to develop this and the author will be experimenting with different techniques. Thus the author decides on using **prototyping** as the Systems development life cycle (SDLC) Model for this project.

3.3.1 Design Methodology

To design the system diagrams for this project Structured System Analysis and Design Modelling (SSADM) methods will be used. SSADM make it easier to design the system iteratively and this complement the choice SDLC method Prototyping.

3.3.2 Evaluation Methodology

Since this is a software framework only performance and functional evaluation can be performed. To evaluate instrumentation components memory and CPU scalability will be used. The Data Science component of this research relies on a convolutional autoencoder which is tasked with the sole purpose of recreating the input to the best of its ability. Due to this nature, only one metric that can be used to evaluate the model is training and testing loss functions (Gondara, 2016).

3.3.3 Requirements Elicitation

As the results of this project will be mostly used by SREs and system administrator the author is hoping to talk with few of the experts in the respective fields to get a better idea on what are the things to be expected from a system like this. Moreover as mentioned in 1.12.2 this system is not designed to entirely replace existing monitoring systems, So the author is hoping to research about production monitoring systems and their workflows to understand how the proposed system could seamlessly integrate them.

3.4 Project Management Methodology

To manage the task of this project, the authors decide to use **Agile PRINCE2**. Agile PRINCE2 is built on the waterfall method which works best for projects with fixed deadlines and requirements with the added benefit of having regulated inputs and outputs (Chappell, 2021).

3.4.1 Deliverables

Deliverable	**Date**
Draft Project Proposal A draft version of this proposal	02nd September 2021
A working beta of MicroSim MicroSim is a tool that simulates a distributed system within a Kubernetes cluster.	15th September 2021
Literature Review Document The Document explaining all the existing tools and published researches on the domain.	21st October 2021
Project Proposal The final version of this project proposal.	04th November 2021
Software Requirement Specification The Document all the key requirements that are gonna get address with this research.	25th November 2021
Proof of Concept Unoptimized prototype with all the main features working.	06th December 2021
Interim Progress Report (IPR) The document explaining all preliminary findings and the current state of the project.	27th January 2022
Project Specifications Design and Prototype (PSDP) A report detailing the initial design and implementation along with a working prototype.	03rd March 2022
Final Project Report Finalize version of the thesis.	5th May 2022

Table 3.2: Deliverables and due dates (self-composed)

3.4.2 Schedule

Gantt chart is a visualisation of tasks with their respective timelines. Refer to Appendix B to find the gantt chart for this project.

3.4.3 Resource Requirement

3.4.3.1 Software Requirements

- **Ubuntu / Arch Linux** - Since this project will use eBPF as a dependency, it will require a Linux kernel-based operating system.
- **Python / R** - Since this project has data science components, relying on a language with a good data science ecosystem will help to develop easier.
- **GoLang / Rust** - While GoLang has an official client library made by Kubernetes developers themselves, the Kube community has developed an excellent alternative in Rust.
- **K3d / Minikube** - To create a Kubernetes cluster locally for development and testing.
- **Intellij / VS Code** - An IDE provides tools to streamline the development process.
- **Latexmk / Overleaf** - Creating the project documentation declaratively will help to keep the formatting consistent. Latex-based editors provide this functionality.
- **Google Drive / Github** - Offsite location to backup the codebase and related documents.
- **ClickUp / Notion** - To manage the project and keep track of things to be done.

3.4.3.2 Hardware Requirements

- **Quad-core CPU with AVX support** - AVX is a CPU instruction set that is optimised for vector operations. Having an AVX-supported CPU could reduce the model inference time.
- **GPU with CUDA support and 2GB or more VRAM** - Both Tensorflow and Pytorch depend on CUDA for hardware-accelerated training.
- **16 GB or more Memory** - Running a microservices simulation locally will consume a lot of memory and while testing models will get loaded into RAM.
- **At least 40GB disk space** - To store the dataset, models docker containers while developing the project.

3.4.3.3 Skill Requirements

- **Experience working with Kubernetes** - The author will be developing a Kubernetes extension so they need to know the inner workings of Kubernetes.

- **Data engineering** - Developing a data encoding technique requires a lot of knowledge on how to manipulate a given dataset.
- **Model engineering** - Creating a model from the ground up is a difficult task. So the author needs to have an in-depth idea about a machine learning framework and how different layers in the model work to fit them properly.

3.4.3.4 Data Requirements

- **Monitoring dataset** - This dataset can be collected using MicroSim tool the author plans to develop to simulate distributed system.

3.4.4 Risk Management

Risk Item	Severity	Frequency	Mitigation Plan
The hypothesis the research is based on is wrong	5	1	Present the findings and explain why the hypothesis was wrong.
Failure in work computer	4	3	Create regular off-site backups.
Lack of domain knowledge	2	3	Talk to a domain expert, keep research.
Models not generalizing	3	4	Explore different methods, try cleaning up the dataset more.
Dataset quality is not up to the standard	4	1	Use a method used in related research to create a new dataset.
Running out of time	1	2	Following a thorough work schedule.
Getting sick and unable to work for few days	3	3	Keeping a few days of a buffer period before deadlines.

Table 3.3: Risks and mitigations (self-composed)

3.5 Chapter Summary

This chapter covered the methodical approach for completing the project so that it is easier to keep track of the end goal and how to get there. Within this, the author's approach to conducting research and development of the minimum viable product was explained. Then, moved on to deliverables and deadlines, finally, this chapter was concluded with requirements and risk assessments to keep the project on track.

4 System Requirements Specification

4.1 Chapter Overview

In this chapter, the author will be talking about the proposed system's requirements and how they were discovered. In initiation, a rich picture diagram will be presented which will explain the high-level overview of the system's interaction with its stakeholders. After that, there will be an in-depth analysis of all positive and negative stakeholders while explaining how each of them views the system. The requirements for the project will be discovered using multiple techniques and both requirement discovery techniques and discovered requirements will be explained within this chapter. Finally, the chapter will be concluded with a context diagram that will explain the system boundaries and a use case diagrams which will be used to visualise the relationships between the functions and the users of the system.

4.2 Rich Picture

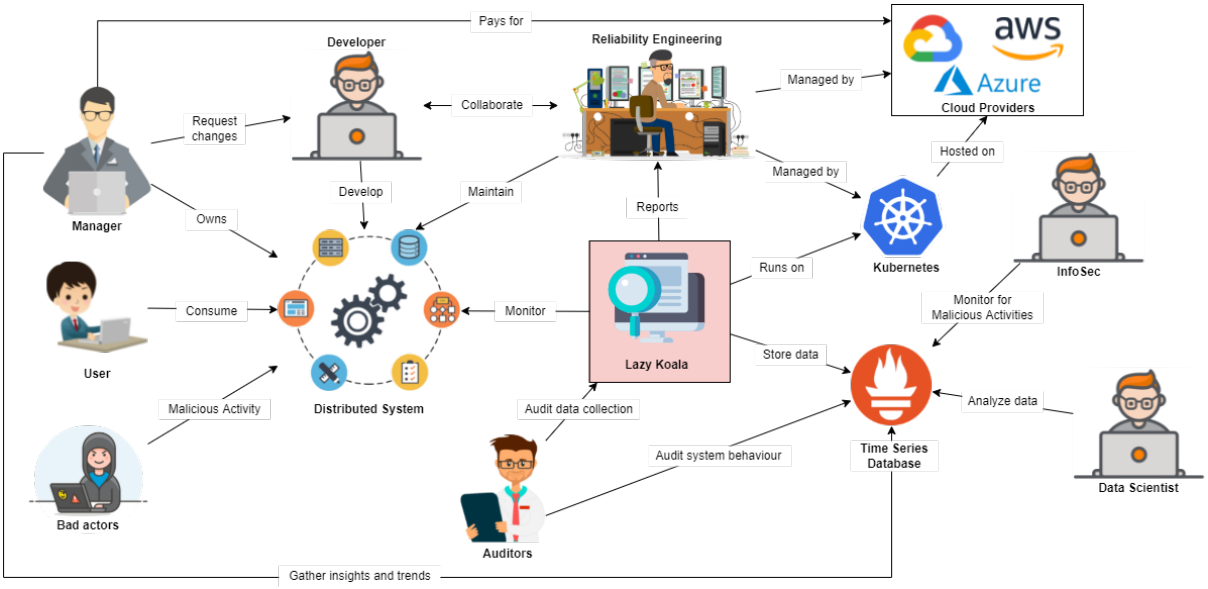


Figure 4.1: Rich picture diagram (self-composed)

The purpose of this project is to create an AI-powered monitoring system that will help SREs detect and diagnose problems in the system quickly. The flow of the system goes as follows; the manager requests the development team to create software to fulfil a customer's need. Then the development team will develop the software and with the help of SREs the software will be deployed for public use. After that, it's SREs duty to keeping the system up and running while doing routine maintenance. While the software is running, a monitoring system will keep an

eye on its health and general behaviour. If one or more components of the system try to deviate from its regular behaviour, it is the monitoring system's job to notify SREs of a few possible reasons for this unexpected behaviour. Finally, SREs and the development team will launch a coordinated effort to identify the real flaw and resolve it as quickly as possible.

4.3 Stakeholder Analysis

To properly plan the system, stakeholders and their roles must be identified. Figure 4.2 shows a visual representation of the stakeholders using the onion model.

4.3.1 Onion Model

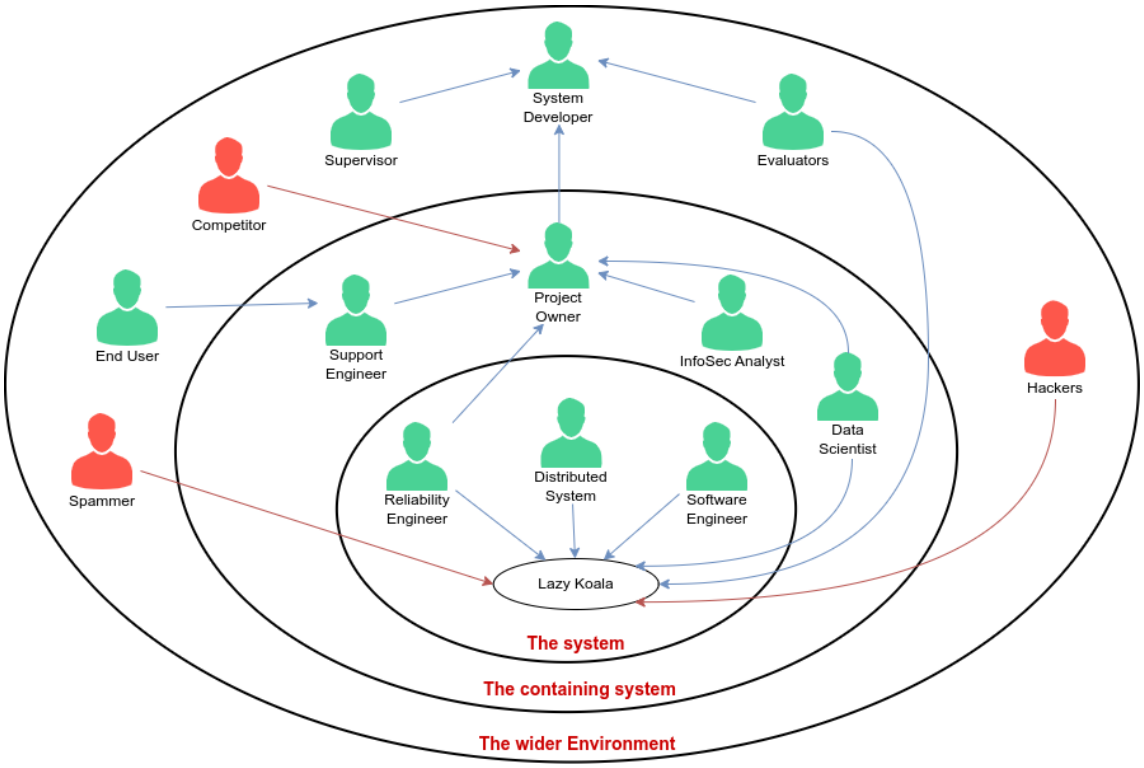


Figure 4.2: Stakeholder onion model (self-composed)

Stakeholder	Role	Viewpoint
Reliability Engineer	Functional Beneficiary	Use the proposed system to understand issues quickly.
Software Engineer	Functional Beneficiary	Use the proposed system to debug issues quickly.
Distributed System	Functional Beneficiary	Become more reliable thanks to lower down time and MTTR.

Project Owner	Financial Beneficiary	Owning a very useful tool that can be licensed to enterprise companies.
System Developer	Financial Beneficiary	Sharpen the development skills and developer portfolio while earning royalty.
Support Engineer	Functional Beneficiary	Less time dealing with frustrated users.
End Users	Functional Beneficiary	Enjoy a more reliable product.
Data Scientist	Functional Beneficiary	Expand upon the concepts and models created for the project.
InfoSec Analyst	Functional Beneficiary	Use the anomaly detection algorithm to find unusual activities.
Evaluator	Advisory	Expand tools and technologies available in the field of reliability engineering.
Supervisor	Advisory	Provide guidance to supervisees so they can successfully complete the project.
Spammer	Negative stakeholder	Creates abnormal behaviors to trigger false alarms and waste developer time and resources.
Hacker	Negative stakeholder	Exploit the proposed system and gain illegal access to the monitored system.
Competitor	Negative stakeholder	Try to replicate results to expand their market share.

Table 4.1: Stakeholder description (self-composed)

4.4 Requirements Elicitation Methodologies

When developing a software project, one of the very first steps that need to be done is requirements engineering. Without following this process, it is difficult to come up with a product that users actually want to use. In this section, the author will describe the techniques he has used to gather requirements with their results.

Literature Review
A Literature review is the fundamental unit of any research project. It helps to understand existing systems and how they work, and also Issues and gaps in those established systems. Since this research project has 3 subcomponents to make up the full system, a literature review was performed on existing instrumentation, anomaly detection, and root cause localisation systems.
Interviews
The target audience for this project will be mostly reliability engineers. Having a one-on-one interview with them and discussing about the project idea and the implementation path uncovered some overlooked use cases and possible improvements to the current implementation.
Self-evaluation
Since the initial idea for the project came from a difficulty the author faced maintaining a distributed system during the industrial placement period. The author was able to perform several self-evaluations during the course of the project and realigned the project scope with the original issue, so the project is always on track.
Brainstorming
As mentioned above, this project originated from a practical problem the author faced. The author was able to use his own experience and self brainstormed some key requirements of the project that he would personally prefer to include.
Prototyping
As the system is getting built, requirements get added and removed due to some requirements getting too complex to build or some additional requirements need to be met to have the core functionality working. This also gives an opportunity to share current progress with a subset of the target audience, get their feedback, and improve upon them.

Table 4.2: Selected requirement elicitation methods (self-composed)

4.5 Analysis of Gathered Data

4.5.1 Literature Review

To find the root course of the failure in a distributed system, three components are required: instrumentation, anomaly detection, and root course localization. A literature review was conducted on each of the components to derive the requirements needed to build an automated root

course analysis platform.

Finding	**Citation**
eBPF is a modern, low overhead technique to extract telemetry by tracing Linux kernel calls.	Molnar (2015)
To find the root course of a fault in a distributed system, three components are required: instrumentation, anomaly detection, and root course localization.	Wu et al. (2020)
Unsupervised learning algorithms excel in identifying unknown patterns.	Silver et al. (2017), Kumarage et al. (2018), Khoshnevisan and Fan (2019)
Convolutional autoencoder network will give the best performance to resources required ratio when it comes to detecting anomalies.	Zhang et al. (2019), Khoshnevisan and Fan (2019)
The root cause could be identified by weighting the anomaly scores in the directed graphs.	Samir and Pahl (2019), Wu et al. (2020), Ma et al. (2020), Meng et al. (2020)
There isn't a common benchmarking or testing platform to test the root causes of algorithms.	Wu et al. (2020), Soldani and Brogi (2021)
Kubernetes is the go to method to manage distributed systems	CNCF (2020)

Table 4.3: Requirements derived from literature review (self-composed)

4.5.2 Interviews

A set of qualitative interviews were conducted to gather external feedback and opinions about the project and proposed system. For this, a senior production engineer, a Senior software engineer, a software engineer, and a trainee DevOps engineer were interviewed. The following table breaks down themes that were emerged after analyzing the transcript and findings based on that.

Theme	**Review**	**Evidence**
Finding the root cause	Almost all the interviewees agreed that whenever there is an outage, it's a cumbersome task to find the root course of that outage. Having a system that makes this experience even slightly better would have huge implications.	”I mean so often, sometimes it's identifying what the problem even is.” - Matthias Rampke (00:31:39) “I mean, figuring out the problem is always hard because you are under stress.” - Jacob Payne (00:23:27)
ML for RCA	This is a bit of a grey area. One of the experts I interviewed said that he had a long history of working with time series forecasting and that results are generally mediocre due to the sheer number of external factors. But since this is a closed system, the results may vary.	“Ideally it helps me understand why AI thinks that this service is the problem.’ - Matthias Rampke (00:35:51) “It could also be useful if it could trace the implications of the outage.” - Jonathan Reiter
Kubernetes	Almost all the technical experts I have talked to agreed that Kubernetes is the way to manage distributed systems in the modern era, and building this system tailor-made to Kubernetes was a wise choice.	“An interesting shift that's been happening in this and is sort of driven by Kubernetes is from tools that imperatively do things to tools that pull a desired state and make it.” - Matthias Rampke (00:25:11) “I mean, in the last five years, obviously Kubernetes blew up. That has been huge for the DevOps community.” - Jacob Payne (00:16:38)

Service Meshes	Experts have a "love-hate" relationship with Service Meshes. Service Meshes offers a lot of utility functions and useful data for debugging. But at the same time, they require a lot of resources to configure and maintain.	"Going through a service mesh is a big investment, so I think there is a space for a smaller thing." - Matthias Rampke (00:05:04)
Proposed Architecture	During interviews, the author showed the Alpha version of the product along with the high-level system diagram, and all the interviewees were impressed by it and excited to get their hands on the final product.	"I'm really looking forward to seeing it in action." - Matthias Rampke (00:11:10) "I can tell you that the project as it is right now has value and if you can get a MVP out there that can be installed as a side car, that's going to be huge." - Jacob Payne (00:36:09)

Table 4.4: Inductive thematic analysis of interviews (self-composed)

4.5.3 Self-evaluation

Criteria	**Finding**
Ways to integrate with Kubernetes	There are two main ways to implement a system that interfaces with Kubernetes. From those creating standalone services that talk to Kubernetes API is the most straightforward method. But during the Google Summer of Code 2021, the author worked on a project which uses a Kubernetes operator framework that is used to extend the functionality of Kubernetes. After evaluating both options, it was decided to rely on the Kubernetes operator framework to build the controller for this project since it is a proven and reliable method to build services that interface with Kubernetes (*The Kubebuilder Book*, n.d.).

Telemetry extraction method	Service meshes are currently the most common way of extracting telemetry, but there is a great focus on eBPF related products due to their efficiency.
New trends in DevOps	From CI/CD to GitOps, DevOps engineers are trying to automate everything that can be automated to minimize the human errors (CNCF, 2020).
Case studies on recent high-profile outages	Few months ago Microsoft experienced a global outage that took more than 24 hours to fully recover from. This waas due to a single bad update to one of their authentication systems (Microsoft, 2021).

Table 4.5: Requirements derived from self-evaluation (self-composed)

4.5.4 Brainstorming

Criteria	**Finding**
Best way to detect anomalies	Learning objective of an autoencoder is given X input, Output X. Even though this doesn't make sense as it is, this process allows the network to deeply understand the underlying function of the given data distribution. So after training at right conditions the model can output a value very close to input resulting in low reconstruction loss. But if the input data is something the model has not seen during the training process (novel anomaly), the model will produce a higher reconstruction loss. This could be used as a signal to identify the health of the service.

Table 4.6: Requirements derived from brainstorming (self-composed)

4.5.5 Prototyping

At the start of this project, a simple Proof of Concept (POC) was developed to understand the feasibility of the project. The experiment started with taking a sine function and concatenating it with a noise function to create a variating data sequence. This was done to emulate service metrics in a small and easy-to-understand way. After that, the author created a simple encoder-

decoder network which was tasked with giving an input sequence of 0 to n to predict the n to n+10 on the sequence. Figure 4.3shows the results of the experiment. The blue line shows the input given to the network, while the green line shows the ground truth results. Finally, the orange line is the model that predicts how the metric should be in any given time step. Notice there is a sudden dip around t=80, it is an artificially injected anomaly that is not present in the training dataset, and due to that, there is a clear difference between expected the current readings from the metric. The author's idea was to use this difference to find anomalies in real-time metrics. Even though this worked well for small-scale prototypes the author found that this does not translate to highly noisy sequences patterns found in production servers but this could be used as the entry point to a more robust solution. Refer to Appendix D for more information on this experiment.

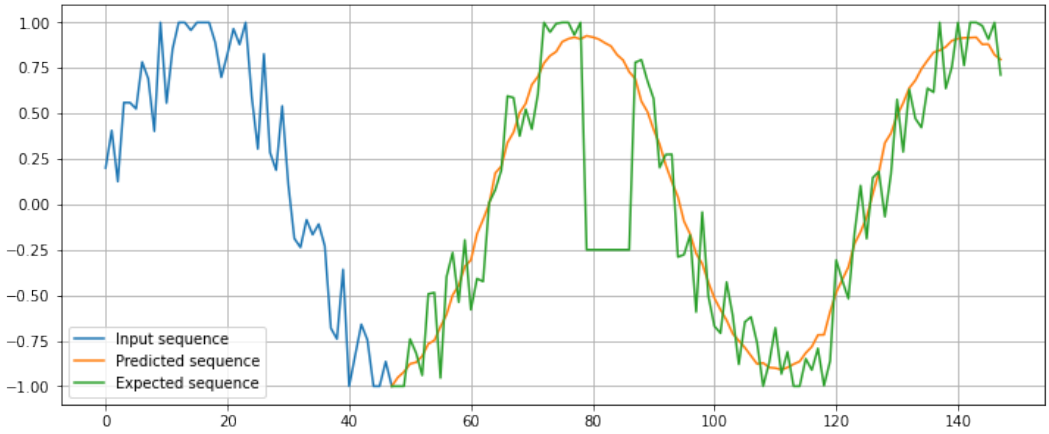


Figure 4.3: The result from the proof of concept (self-composed)

4.6 Summary of Findings

Findings	Literature Review	Interviews	Self-evaluation	Brainstorming	Prototyping
Finding the root cause of a problem during an outage is a time consuming task.		X	X		
It is possible to use machine learning to find anomalies from time series data.	X				X
There is a great trend towards "automating boring tasks".		X	X		

Lots of companies are looking to migrate from monolithic architecture to microservices.	X	X			
Kubernetes is the most popular way to manage distributed systems.	X	X	X		
eBPF provides a low overhead method to collect telemetry data without additional instrumentation.	X		X		X
Autoencoder are good at finding and forecasting patterns in data.	X			X	X
There aren't any established methods to test monitoring systems, Creating a testing toolkit for that would help future researchers.	X	X			
Having a dashboard which could show a blast radius of a system failure would help to reduce the MTTR.		X	X		
The Kubernetes operator framework is the best way to build Kubernetes native applications.		X		X	X

Table 4.7: Summary of findings (self-composed)

4.7 Context Diagram

Since the proposed system falls into the category of an open system, it communicates with a lot of external parties. It's advised to have a clear system boundary when developing such a system so the development process does not get overwhelming. Figure 4.4 explains the main interactions between the system and the third parties.

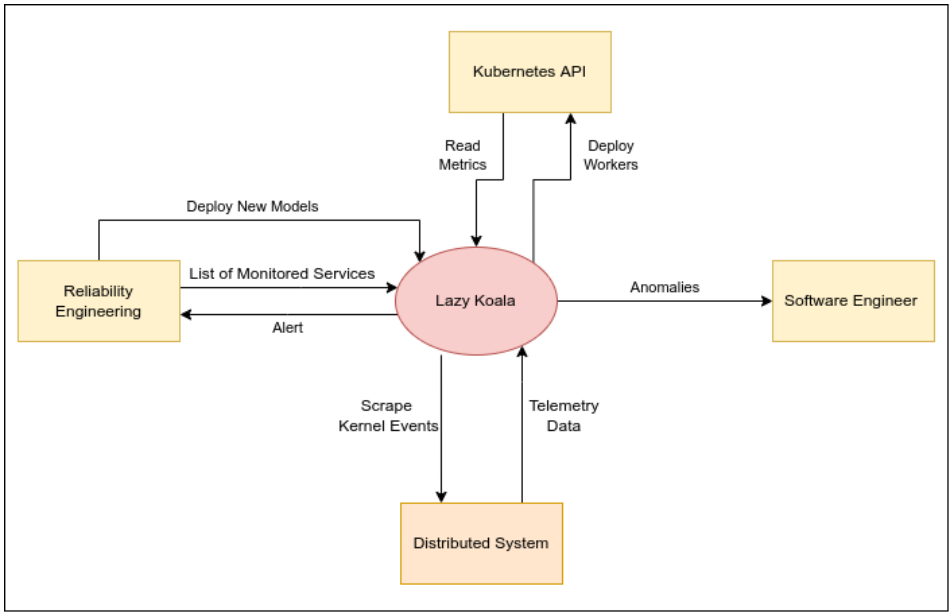


Figure 4.4: Context diagram (self-composed)

4.8 Use Case Diagram

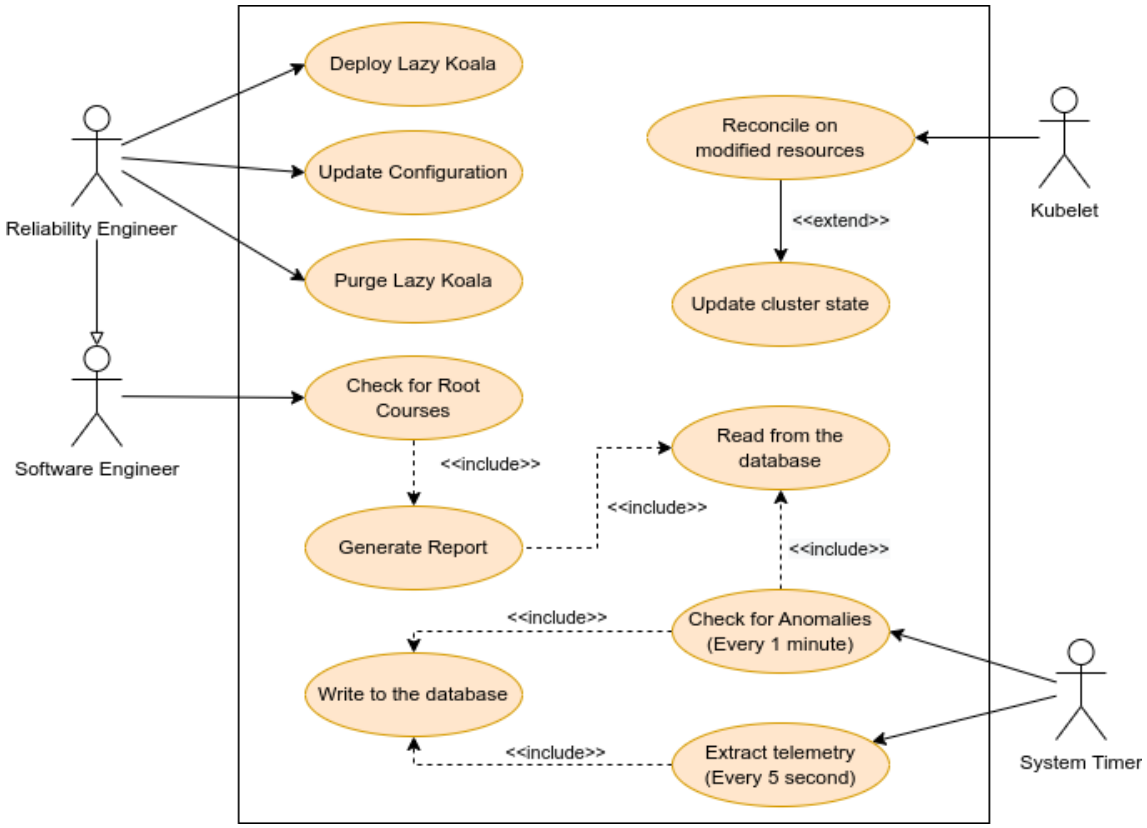


Figure 4.5: Use case diagram (self-composed)

4.9 Use Case Descriptions

Due to the page limits, only the main use-case description is present here. Please find reference of the use-case descriptions in Appendix-C.

Use Case ID	**UC-04**
Use Case Name	Check for Root Courses
Description	Look at the service topology graph to find out the root course of an issue.
Participating actors	Software Engineer Reliability Engineer
Preconditions	• kubectl installed and configured to talk to a Kubernetes cluster. • The Kubernetes cluster has the Operator deployed. • Establish a port forwarding connection with Operator.
Extended use cases	N/A

Included use cases	Generate Report Read from the database
Main flow	1. Visit the forwarded port on the local machine. 2. Open Monitor tab. 3. Inspect the graph.
Alternative flows	N/A
Exceptional flows	**E1**: the Operator returns non 200 status code. 1. Show the error to the user.
Postconditions	N/A

4.10 Requirements Specifications

Since this project touched the deep ends of both SRE and data science expectations. The priorities of the system must be managed to achieve a reliable and functioning system at the end of the deadline. Therefore, to achieve this, the MoSCoW prioritization model was used.

Priority Level	Description
Must have	Requirements which need to be met to have minimum viable product.
Should have	Requirements that need to be completed to have a usable product.
Could have	Nice to have requirements that would improve the quality of life of the system.
Will not have	Requirements that will not get covered during this iteration but might get implemented in the future.

Table 4.9: Requirement priorities (self-composed)

Use Case ID	Use Case Name
UC-01	Deploy Lazy Koala
UC-02	Update Configuration
UC-03	Purge Lazy Koala
UC-04	Check for Root Courses
UC-05	Generate Report
UC-06	Read from the database
UC-07	Extract telemetry (Every 5 second)

UC-08	Check for Anomalies (Every 1 minute)
UC-09	Write to the database
UC-10	Reconcile on modified resources
UC-11	Update cluster state

Table 4.10: Use cases of the system (self-composed)

4.10.1 Functional Requirements

ID	Requirement and Description	Priority Level	Use case
FR01	**Users should be able to deploy the Operator to an existing Kubernetes cluster.** Operator should work on any Linux-based Kubernetes cluster with version 1.22 or higher without any additional configuration from the user's end.	Must have	UC-01
FR02	**Users should be able to remove the Operator completely from the cluster.** Once uninstalled all the provisioned resources should be cleaned up by Operator itself.	Should have	UC-03
FR03	**Users should be able to specify which services need to be monitored.** System should allow the user to exclude some services getting tracked.	Must have	UC-02
FR04	**Users should be able to see the services monitored by Operator** System should be transparent to the user about the monitored and unmonitored service.	Could have	UC-05
FR05	**Operator should deploy an instance of Telemetry extraction agent (Gazer) to every node in the cluster.** In the Kubernetes cluster every node has a separate instance of the Linux kernel. So for every instance of Linux kernel, an instance of Gazer must be present to ensure all the relevant data is captured.	Must have	UC-01, UC-10

FR06	**Gazer should intersect all "inet_sock_set_state" kernel calls and export the relevant data to Prometheus.** Whenever a userspace application makes a TCP call, this kernel method is invoked to communicate with the network interface. Inspecting the data structures of this will allow us to extract a lot of information about each TCP calls.	Must have	UC-07, UC-09
FR07	**Gazer periodically poll the size of "sk_ack_backlog" for interested ports to export the relevant data to Prometheus.** This kernel data structure holds the TCP connections that are left to be acknowledged. Knowing the size of this queue will help to understand the efficiency of each service.	Should have	UC-07, UC-09
FR08	**Gazer should poll the Kubernetes metric server periodically and export the relevant data to Prometheus.** Sudden changes in CPU and Memory usage will be a good indication of an anomaly. Therefore, exporting those to be processed later will be wise.	Must have	UC-07, UC-09
FR09	**Operator should periodically check for changes in monitored services and update the Gazer ConfigMap.** Kubelet is watching over all the services on the system and restarts them if they become unhealthy. With that, the IP address of that service is gonna change and Operator is responsible to let Gazer know about such changes.	Should have	UC-10, UC-11
FR10	**Gazer should react to config updates in realtime.** When a Operator pushes a new config, Gazer should look for the new IPs without requiring a complete reset. Which is time consuming and expensive.	Could have	UC-10, UC-11
FR11	**AI-engine (Sherlock) should provision the list of models given by Operator** Sherlock should react to updates from Operator and provide models on demand.	Must have	UC-10

FR12	**Sherlock should periodically calculate the anomaly score for each of the monitored services and export it to Prometheus.** Anomaly score is used by the UI to understand the spread of an anomaly.	Must have	UC-08, UC-09
FR13	**Operator should have a Web UI to visualize the service topology.** UI should help users to visualize the spread of an anomaly throughout all of the monitored service.	Should have	UC-04, UC-05, UC-06
FR14	**Operator should add a finalizer for each of the provisioned resources.** Finalizers ensure the parent of a resource will not be deleted before all children are cleaned up. This avoids leaving the cluster with orphaned resources that won't be cleaned up without user intervention.	Should have	UC-03
FR15	**Operator should periodically fine-tune models.** Microservices could get complex and change over time. To combat this Operator make sure that the models will adapt to the new changes done in monitored services.	Will not have	UC-04, UC-10

Table 4.11: Functional requirements (self-composed)

4.10.2 Non-Functional Requirements

ID	Description	Specification	Priority Level
NFR1	Operator should follow Principle of Least Privilege when accessing Kubernetes APIs.	Security	Must have
NFR2	Systems should have a fragmented architecture so that each component can be individually scaled in order to save resources.	Scalability	Must have
NFR3	Each component should work individually such that users can install parts of the system they are interested in.	Usability	Could have

NFR4	Gazer should be limited to using only 100 mCPUs and 80MB of memory.	Performance	Should have
NFR5	Sherlock should be limited to using only 100 mCPUs and 100MB of memory.	Performance	Could have
NFR6	Operator should be packaged as a Helm Chart for ease of use.	Usability	Must have
NFR7	Reconstruction error of Sherlock should be under 0.85	Performance	Could have
NFR8	Operator's reconciliation loop should use the exponential backoff technique when there is an error while reconciling for a configuration change.	Reliability	Must have
NFR9	Follow Coding best practices and rely on linters for code formatting.	Maintainability	Could have
NFR10	The project should be backed by an automated CI/CD tool to test and build each component with each release.	Maintainability	Could have

Table 4.12: Non-Functional requirements (self-composed)

4.11 Chapter Summary

This chapter started by explaining the stakeholders and their involvement in this project. Then proceeded to explain the qualitative requirements engineering techniques used to identify the requirement of this project along with findings from each of the techniques. Next, the flow of information within the system explained using a context diagram. Then, use cases of the system derived from identified requirements, and main use cases further explained with use case descriptions. Finally, the chapter concluded with an overview of both functional and non-functional requirements along with their priority levels.

5 SLEP Issues and Mitigation

5.1 Chapter Overview

This chapter talks about how social, legal, ethical, and professional issues affect this project and steps taken to mitigate them.

5.2 SLEP Issues and Mitigation

Social	**Legal**
Throughout the both requirement elicitation phase and the evaluation phase, several interviews have been carried out. The interviewee's name was included to provide credibility for the author's claims. The author reached out to them and obtained their written consent and only included the names of those who agreed to publish their name in this research project.	Several open-source libraries and tools were used to simplify the development. All the licenses that came with those, were properly cited within the code base along with references to original sources. Finally, it was decided to publicize this project under the MIT licenses as a token of gratitude to the FOSS community.
Ethical	**Professional**
One of the most prominent ethical issues that rise during the research process is plagiarism. All the literature which was used to compile this document was cited properly to prevent this. On the implementation side, even the smallest code snippets that were taken from a third-party source were properly cited to the original sources in order to provide the credit where credits are due.	Since this project is meant to be extended by other researchers, implementation was done following software engineering best practices such as code commenting and generalizations. Additionally, the testing and the evaluation of the project were done with a system that closely resembles the real-world scenarios and the collected data was presented without any manipulation.

Table 5.1: SLEP issues and mitigations (self-composed)

5.3 Chapter Summary

The above chapter described the participation of social, legal, ethical and professional issues in this project and how those were mitigated to comply with the BCS code of conduct.

6 System Design

6.1 Chapter Overview

This chapter focuses on the overall design of the proposed system. First, the author will discuss the design goals in creating this system and the philosophies behind it. Then the system architecture will be explained and how each layer of the system integrates with each other. Finally, the chapter will conclude with the design of the proposed system along with the low-fidelity UI design, which will showcase how the UIs of the system will look.

6.2 Design Goals

Design Goal	Description
Modularity	Since this is designed to work in a cloud-native environment, it is considered best practise to have all components loosely coupled. During the requirement engineering phase two of the industry experts expressed their interest to integrate this project into some of their existing toolings. So having modular design will help their efforts too.
Lightweight	As this was designed to be a supporting system to existing distributed systems, it needs to be as lightweight as possible to justify its use of this. If the supporting system consumes more resources than the target system, it will not be practical to use.
No Code Change	It's unlikely for developers to update all the services in a distributed system to match with a monitoring system. Therefore, to increase adaptability, this system should be able to work without any instrumentation from the developers' side.
Extensibility	One of the core goals of this project is to be a starting place for future researchers who are looking into root cause analysis. So having this toolkit extensible will greatly help their efforts.
Scalability	Since the main target audience of this product is large enterprises with huge systems, this system should be able to scale up to their level in order to be relevant.

Table 6.1: Project design goals (self-composed)

6.3 System Architecture

System architecture design gives a bird's-eye view of how all the components in the system communicate with each other. This helps us to understand the dependencies and responsibilities of each component. Since this system is designed to run on a microservices-based environment, an n-tier design architecture was used to physically separate the components in the system to have better a reliability and scalability.

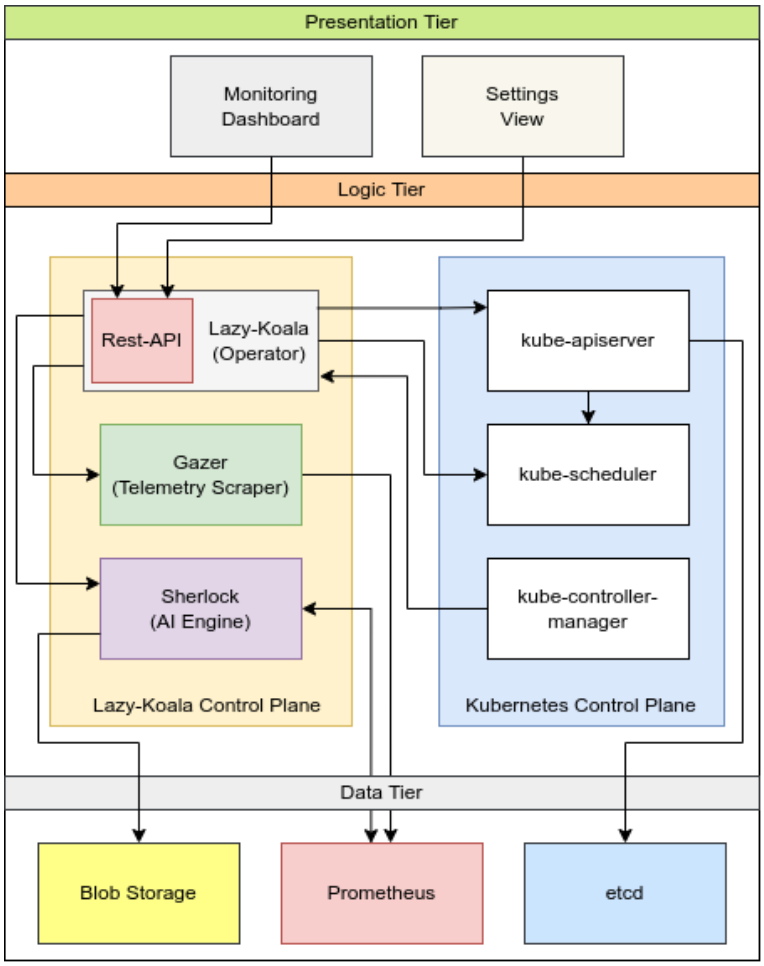


Figure 6.1: Tiered architecture (self-composed))

6.3.1 Presentation Tier

The presentation tier will be entirely running in the client's computer while depending on the logic tier for data.

- **Monitoring Dashboard** - This view is responsible for helping the user understand the service topology and visually identify issues in the system.
- **Settings View** - On the settings page users can choose which services are needed to be monitored along with their DNS address.

6.3.2 Logic Tier

The logic tier will contain three custom microservices that depend on Kubernetes's core modules to operate.

- **Operator** - The Operator is the main bridge between Kubernetes APIs and this system. It also contains a proxy server that redirects incoming client requests to kube-api.
- **Gazer** - An instance of Gazer will be running on every node in the Kubernetes cluster which passively extracts telemetry and sends them over to the Prometheus server.
- **Sherlock** - The AI engine periodically queries Prometheus to get the current status of all the monitored services. Then it calculates an anomaly score for each service and pushes it to Prometheus so it can be sent back to the presentation layer
- **kube-apiserver** - This is an API provided by Kubernetes that help to read and update the cluster status programmatically.
- **kube-scheduler** - kube-scheduler is responsible for smartly provision requested resources in available spaces.
- **kube-controller-manager** - This service sends updates to all operators running in the cluster whenever there is a change to a resource that was owned by the specific operator.

6.3.3 Data Tier

- **Blob Storage** - Storage for the pre-trained models and built containers.
- **Prometheus** - A time series database that is highly optimised for telemetry collection.
- **etcd** - etcd is an in-memory database that will be responsible for holding the resources specifications and Gazer config.

6.4 System Design

6.4.1 Design Paradigm

When building a software application, there are two main design paradigms to choose from to organize the codebase. Object-Oriented Analysis and Design (OOAD) which is very popular among programming languages such as Java and C# is a way of mimicking the behavior of real-world objects and how they interact in the real world. However, this project has a lot of components that are loosely coupled and implemented in various languages and frameworks. Therefore, Structured Systems Analysis and Design (SSADM) was chosen as the design paradigm for this project.

6.4.2 Data-flow diagram

The Data-flow diagram explains the flow of request data within the system and how each process in the system interacts with each other at a higher level.

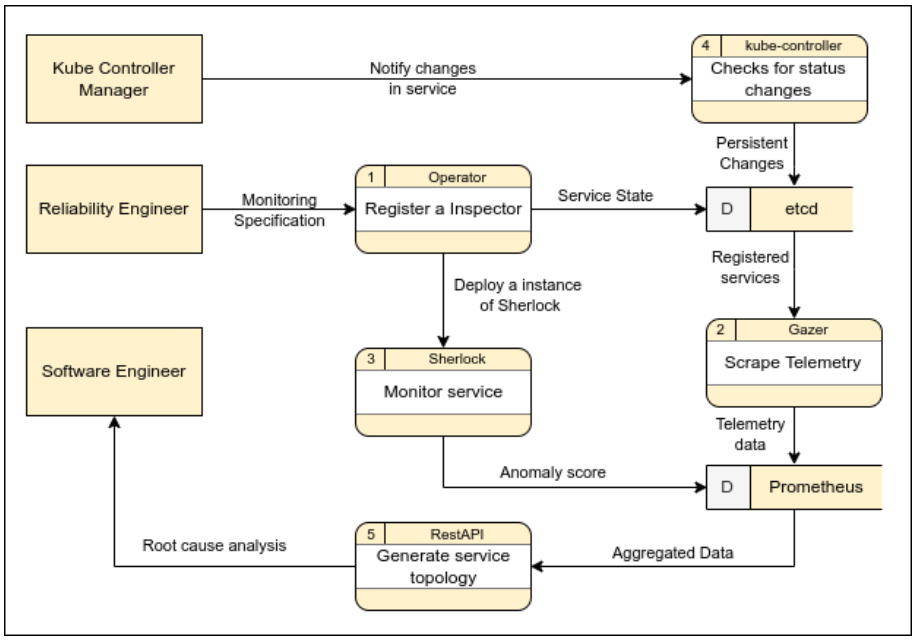


Figure 6.2: Data-flow diagram - level 1 (self-composed)

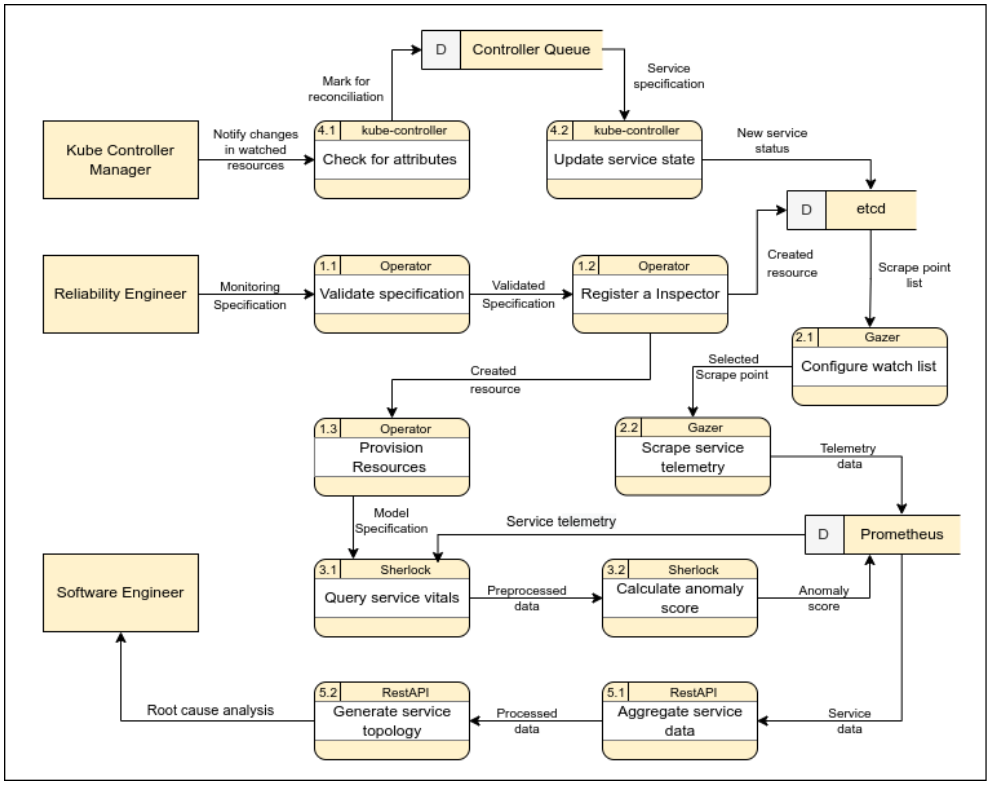
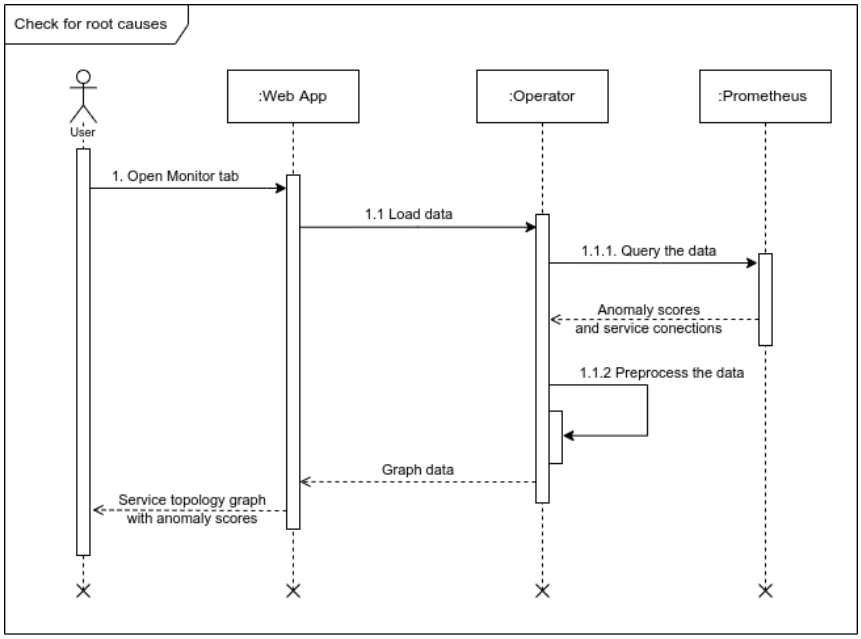


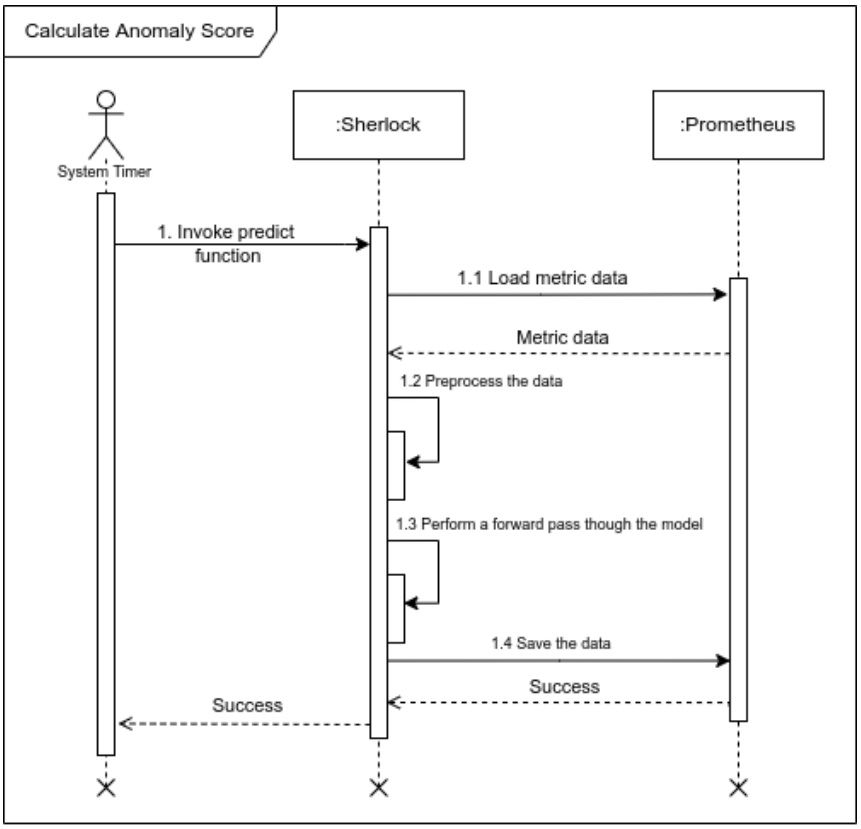
Figure 6.3: Data-flow diagram - level 2 (self-composed)

6.4.3 Sequence Diagram

Sequence diagrams are meant to showcase the flow of instructions within sub-components of the system. The following diagram explains how the system reacts when two of the main core functionality are invoked.



(a) Check for root cause



(b) Calculate anomaly score

Figure 6.4: Sequence diagrams (self-composed)

6.4.4 System Process Flow Chart

The process flow chart describes how the data flow within the system along with decisions that are made to control the flow. The following diagram shows how the system handles the process of adding a new service to the monitored service list.

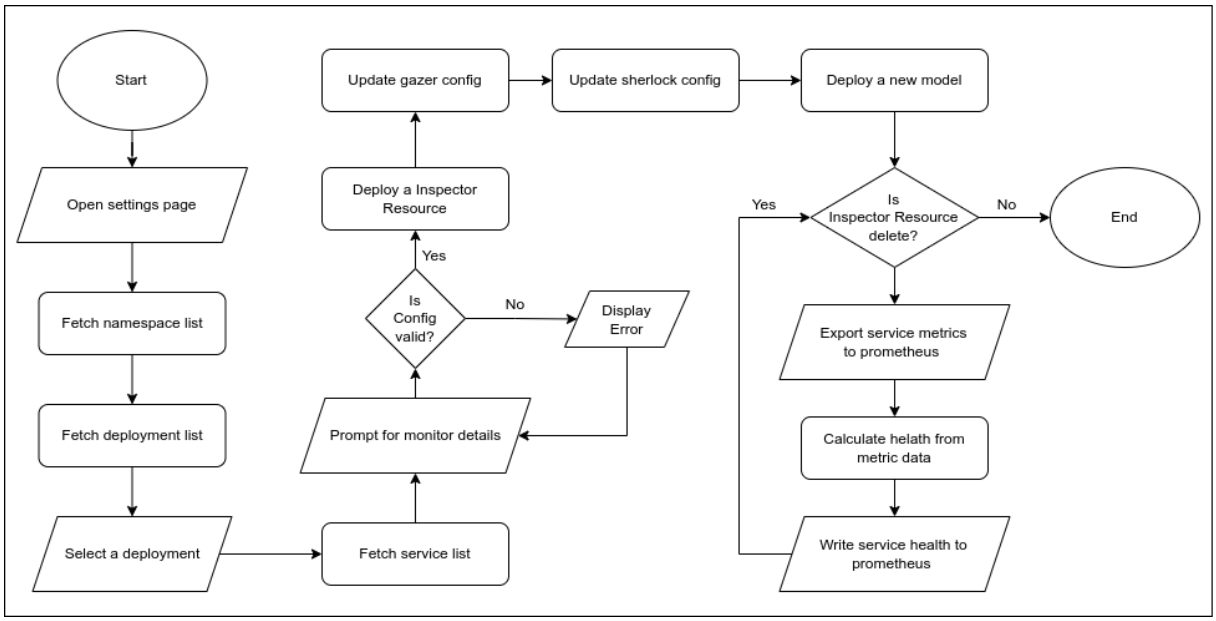
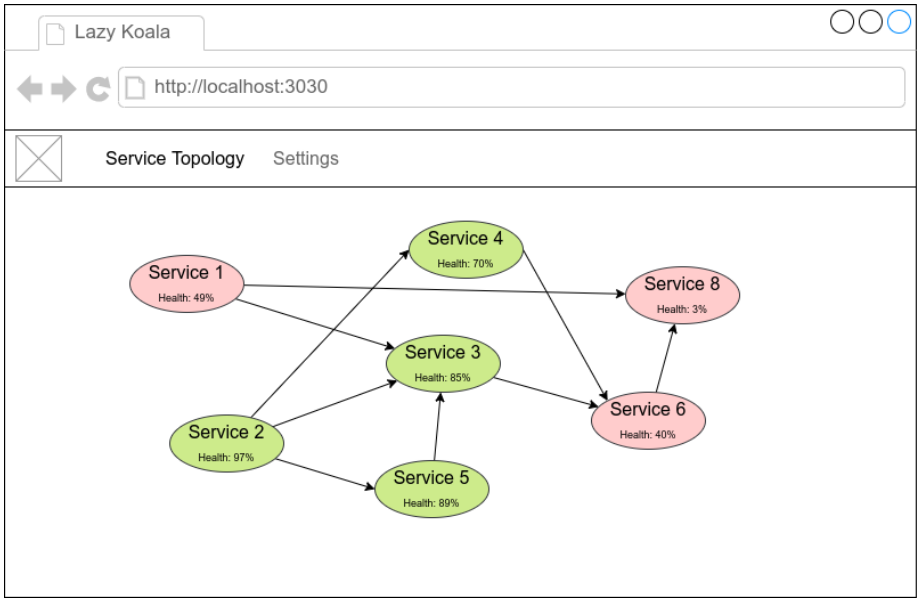


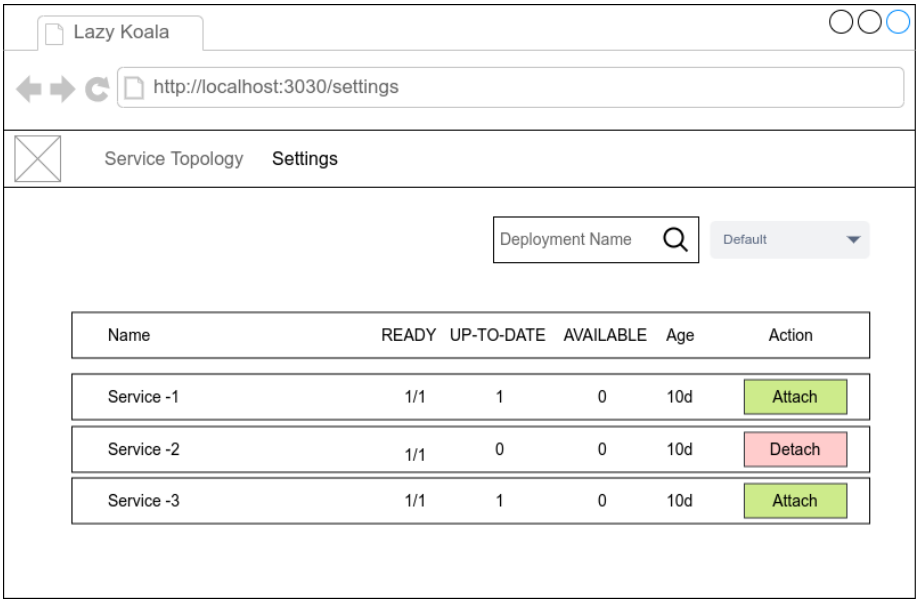
Figure 6.5: Process flow chart (self-composed)

6.4.5 UI Design

Since this project was developed as a Kubernetes native application, most of the functionality works as a daemon process in the background. However, there are two use cases where having a visual user interface greatly increases the usability of this project. UI mockups attached below showcase two of those use cases. Figure 6.6a depicts how developers will be able to inspect the topology of the system and find issues in real time, while Figure 6.6b showcases the settings page which is used to tag interested services in the system which needs to be monitored. The high-fidelity version of these digrams can be found in Appendix G



(a) Inspector View



(b) Settings View

Figure 6.6: UI mockups (self-composed)

6.5 Chapter Summary

After discovering requirements through many methods, this chapter focusses on creating a system around those requirements. The chapter started with design goals the author has set for himself and moved over to explain all the layers of the system in detail with each component. Finally, the chapter was concluded with an explanation of the design paradigm and diagrams associated with it.

7 Implementation

7.1 Chapter Overview

This chapter focusses on making the proposed system a reality. During this chapter, the author will discuss the tools and technologies he relied on to complete the working prototype along with the reasoning behind all those choices. Then the author will share their experience implementing the core functionality of the system in line with his design goals. Finally, the chapter will be concluded with a self-reflection of the achievements.

7.2 Technology Selection

7.2.1 Technology Stack

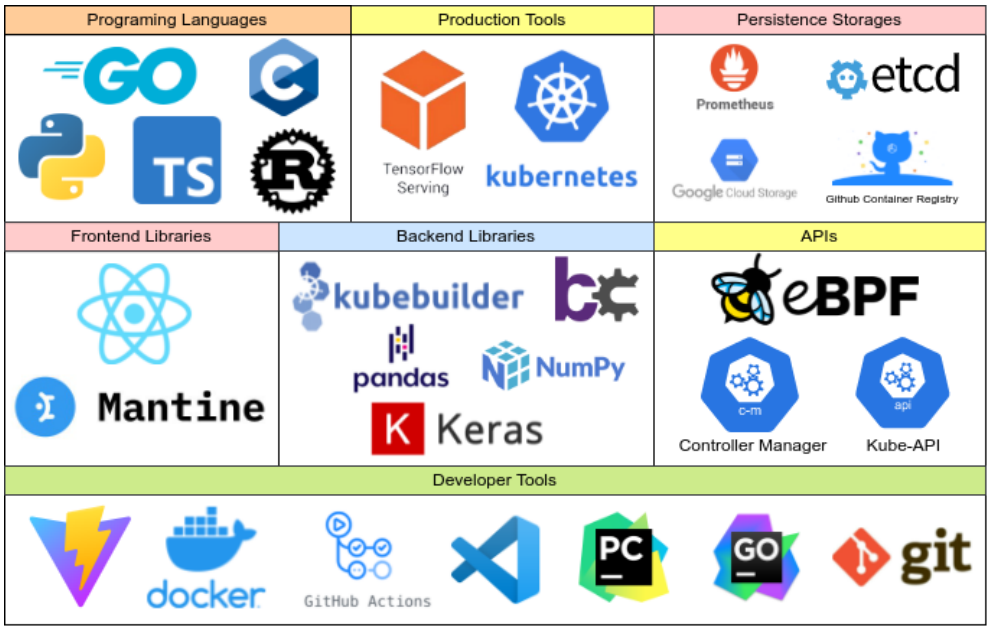


Figure 7.1: Technology stack (self-composed)

At a grasp, this seems like a lot of tooling for a project of undergraduate level, but each and every tool and technology listed here provides vital functionality to make the final prototype efficient and reliable as much as possible in order to stand through to the design goals listed above.

7.2.2 Programming Language

This project was built using five different programming languages which were chosen due to their unique features that are required from different components in the system.

- **GoLang** - Go is a programming language invented by Google that took a lot of inspiration from C, but with modern features like memory safety, automatic garbage collection, and

built-in concurrency. Due to this nature Go was used to build Kubernetes itself and almost all the tooling around Kubernetes relies on as their language of choice. Since this project tries to extend some functionality of Kubernetes, it was highly recommended to use Go to build parts of the system that interface with the Kubernetes tooling.

- **Python** - Python is known as the go-to language for data science but in this project, Python had another critical role. There is a library called BCC that streamlines the connection to the Linux eBPF API. So both data science and telemetry extraction components are built using Python.
- **C** - Linux eBPF API allows userspace applications to submit some sandboxed programs into kernel space at runtime, and the kernel is responsible for compiling and executing them along with kernel calls. But since this is a Linux kernel feature, all sandboxed programs need to be written in C so the kernel can understand it.
- **TypeScript** - For frontend applications, programming languages are limited to Javascript and TypeScript. TypeScript was chosen for this project, since TypeScript offers a lot of compile-time checks which prevent accidental bugs from sneaking into the production application.
- **Rust** - Out of all tools and technologies this may be the only replaceable technology. Rust will be used in the Sherlock module to interface with the machine learning model. Even though Python or even Go could be viable alternatives to this, Rust has the lowest memory and CPU footprint of any modern programming language. Since one of the design goals of this project is to have the lowest overhead possible the author was settled on relying on Rust for this task.

7.2.3 Libraries Utilized

Software libraries prevent software engineers from reinventing the wheel every time they want to perform some common functionality. This is done by empowering them with abstractions that they can use to perform the task they wanted so that they can focus on important things. To build this project, several libraries were used in both the UI and the back-end components.

7.2.3.1 Frontend

- **ReactJS** - Since this is a hosted application a web-based UI made more sense. In order to build a web-based front-end, there are a few common methods; it is possible to use a vanilla HTML stack and build everything from scratch, but for this project, UIs need

to be interactive, and that left the author with three viable choices, ReactJS, VueJS, and Svelte. Since the author was more familiar with ReactJS, he chose to rely on it for the UI implementation.

- **Mantine** - Mantine is a React component library which helps developers to use the prebuilt component like date pickers without having to code them from scratch. Viable alternatives for these are Boostrap, MaterialUI, and Ant Design. The author settled on Mantine due to its user-friendliness and rich TypeScript support.
- **Force Graph** - force-graph is a UI library that helps to create interactive graphs. This library was used to create the service topology graph. There weren't any alternatives for this that allowed to render directed graphs with interactivity.

7.2.3.2 Backend

- **Kubebuilder** - Kubebuilder is a SDK that is used to build Custom Kubernetes Operators. This is created and maintained by the Kubernetes special interest group themself along with the community support. The only viable alternative for this is called Operator SDK, which is developed by the community with the support of RedHat. Since the author had prior experience with Kubebuilder and its inner working, it was decided to rely upon it.
- **BCC** - BCC is a front-end to Linux eBPF API which makes deploying kernel probes and retrieving data from them very easy. In addition to relying on C interfaces, this was the only viable solution.
- **Pandas** - Pandas is a data manipulation and an analysis library that was used in both Gazer and Sherlock module.
- **Numpy** - Is considered a holy grail when it comes to data science work in Python. Even libraries like Tensorflow and PyTorch rely on this for manipulation of multidimensional arrays and matrices. In the training phase of Sherlock, NumPy was extensively used to preprocess the datasets.
- **Keras** - Keras is an abstraction on top of Tensorflow that helps to create a deep learning model with minimum boilerplate code.

7.2.4 Persistence Storages

- **Prometheus** - Prometheus is a time series database that doubles as a data scraping agent. Prometheus uses the pull method in contrast to normal pushing methods to update the database. During the requirements engineering phase, it was discovered that the vast

majority of companies rely on Prometheus.

- **etcd** - etcd is a key-value data store which is built into the core of Kubernetes. The Operator relies on this to sync up the monitoring configuration with Gazer instances and to keep track of all the monitored services.
- **Github Container Registry** - This stores all the built containers, so that the Kubernetes cluster can easily pull down spin copies of them. Eventhough there are alternatives like DockerHub, Github Container Registry was used due to the tight integration with Github.

7.2.5 Developer Tools Utilized

- **Vite** - A JavaScript build tool that can convert TypeScript code into well-optimized JavaScript. Vite was selected over webpack because of its efficiency in build times.
- **Docker** - A container management tool that can package software into a lightweight self-containing package that can be deployed in Kubernetes.
- **Github Actions** - Automated CI/CD platform which is built into Github platform.
- **VSCode** - Code editor used to create frontend UIs.
- **PyCharm** - IDE used to develop Gazer and Sherlock.
- **GoLand** - IDE used to develop the Operator.
- **Git** - Version control tool that was used to keep track of changes between releases.

7.2.6 Production Tools

- **Tensorflow Servings** - Production grade machine learning model serving system written in C++ to be as efficient as possible.
- **Kubernetes** - Host the entire system along with a distributed system that's get monitored.

7.2.7 Summary of Technology Selection

Component	**Tools/Technologies Used**
Operator	Go, etcd, Kubebuilder, Kube-API, Controller Manager, GoLand
Gazer	C, Python, Prometheus, BCC, Pandas, eBPF, Kube-API, PyCharm
Sherlock - Training Phase	Python, Prometheus, Pandas, Numpy, Keras, PyCharm
Sherlock - Production	Rust, Prometheus, TF-Servings, Github Container Registry
User Interface	TypeScript, React, Mantine, D3.js, Vite, VSCode
Common for all	Docker, Github Actions, Git, Kubernetes

Table 7.1: Summary of technology selection (self-composed)

7.3 Implementation of Core Functionalities

This project contains three components that work together to make up the entire system. In this section the inner working of each of those components will be explained.

7.3.1 Lazy Koala Resource Manager (Operator)

Operator is the heart of the entire project. It is responsible for binding all other components together. In the context of Kubernetes, an operator is an agent that is running on the cluster, which is responsible for keeping one or more dedicated resources in sync with the desired state.

For example, Kubernetes has a built-in resource named ”Pod” which is the smallest deployable object in Kubernetes. So when a system administrator asked the Kube-API to create a pod out of a certain Docker container, Kube-API will create a resource object and attach it to the pod operator. Once that is done, the pod operator will parse the pod resource specification and create a pod out of it. If for some reason the pod crashes or the administrator changes the specification of the pod, the operator will be notified, and it will rerun its reconciliation function to match the observed state with the desired state.

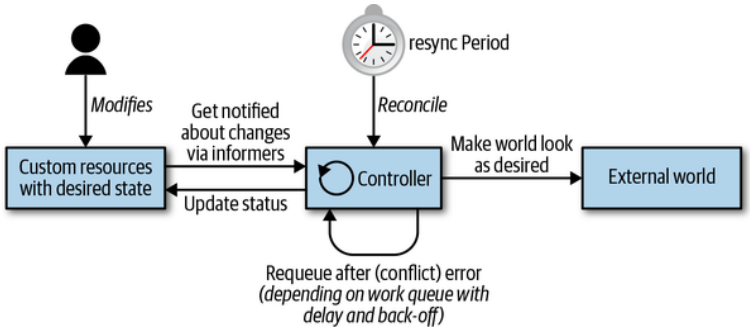


Figure 7.2: Kubernetes control loop (Hausenblas and Schimanski, 2019)

Returning to Operator, it has a Custom Resource Definition (CRD) called Inspector. In its specification, there are three required values. Deployment reference, DNS reference, and an URL to download the model that was fine-tuned for this specific deployment. So once a resource is deployed, the Operator will first get the pods related to the deployment and populate the ”scrapePoints” data structure with the IP address of each pod. Then it is going to find the IP address mapped to the DNS reference and append that to the ”scrapePoints”. Then Operator will compare the new scrapePoints hashmap with the scrapePoints hashmap created in the previous iteration. Then it will identify that points needs to be added and which needs to be removed from the ”gazer-config”. After that the Operator will pull down the ”gazer-config” ConfigMap

and run the process through the calculated changelog. Then it will send a patch request to the kube-api with the new state. Since this ConfigMap gets mounted to every Gazer instance via Kubernetes volumes system the changes made here will instantly be reflected with all of the Gazer instances. As a final step, an instance of the specified model will be provisioned within the tensorflow servings container which is running inside Sherlock deployment.

Figure 7.3 shows a part of this reconciliation loop, which get repeated every time there is a change to an existing resource or whenever the user creates a new variant of this resource.

```
func (r *InspectorReconciler) Reconcile(ctx context.Context, req ctrl.Request) (ctrl.Result, error) {
    ....
    ....

    // Get the intended deployment
    var deploymentRef appsv1.Deployment
    if err := r.Get(ctx, types.NamespacedName{
        Namespace: inspector.Spec.Namespace,
        Name:      inspector.Spec.DeploymentRef,
    }, &deploymentRef); err != nil {
        return ctrl.Result{}, err
    }

    scrapePoints := make(map[string]ScrapePoint)

    // Get Pods for that deployment
    selector := client.MatchingLabels(deploymentRef.Spec.Selector.MatchLabels)
    inspector.Status.PodsSelector = selector
    var podList v1.PodList
    if err := r.List(ctx, &podList, &selector); client.IgnoreNotFound(err) != nil {
        logger.Error(err, fmt.Sprintf("failed to pods for deployment %s", deploymentRef.ObjectMeta.Name))
        return ctrl.Result{}, err
    }

    // Create Scrape point for each pod
    for _, pod := range podList.Items {
        scrapePoints[pod.Status.PodIP] = ScrapePoint{
            Name:        pod.ObjectMeta.Name,
            ServiceName: inspector.ObjectMeta.Name,
            Namespace:   inspector.Spec.Namespace,
            Node:        &pod.Spec.NodeName,
            IsService:   false,
        }
    }

    ....
    ....

    // Patch the config file
    configMap.Data["config.yaml"] = string(encodedConfig)
    if err := r.Update(ctx, &configMap); err != nil {
        return ctrl.Result{}, err
    }

    ....
    ....

    return ctrl.Result{RequeueAfter: time.Minute}, nil
}
```

Figure 7.3: Operator reconciliation loop (self-composed)

7.3.2 Telemetry extraction agent (Gazer)

Gazer is the telemetry extraction agent that is scheduled to run on every node on the cluster using a Kubernetes DaemonSet. Gazer is implemented in Python with the help of a library called BCC which acts as a front-end for the eBPF API. Gazer contains two kernel probes that are submitted to the kernel space at the beginning.

The first probe is a TCP SYN backlog monitor that keeps track of the backlog size

of the TCP SYN queue. Since every TCP connection starts with a 3-way handshake, the SYN packet is the first packet that will be sent from the client in this sequence. The entire request is held until this packet is acknowledged by the system. Therefore, an unusually high SYN backlog is a strong signal of something going wrong. Figure 7.4 shows the core part of this probe and how the size of the backlog is calculated.

```
// https://www.kernel.org/doc/htmldocs/networking/API-struct-sock.html
int kretprobe__tcp_v4_syn_recv_sock(struct pt_regs *ctx){
    struct sock *newsk = (struct sock *)PT_REGS_RC(ctx);

    ....
    ....

    struct sock *skp = *skpp;
    // Update the histogram
    backlog_key_t key = {};
    key.backlog = skp->sk_max_ack_backlog;
    key.saddr = newsk->__sk_common.skc_rcv_saddr;
    key.lport = newsk->__sk_common.skc_num;
    key.slot = bpf_log2l(skp->sk_ack_backlog);
    syn_backlog.atomic_increment(key);
    return 0;
}
```

Figure 7.4: eBPF probe to collecting tcp backlog (self-composed)

Next is a tracepoint probe that gets invoked whenever inet_sock_set_state kernel function is called. This probe extract five key data points from every TCP request. The IP address that is transmitted and received, the number of bytes sent and received, and finally the time taken to complete the entire request. All these data are shipped to the userspace via a perf buffer. In the user space, these raw data are enriched with the data collected from Kube-api.

As shown in the figure 7.5, since Gazer already has a list of interested IP addresses which is given by the Operator, it first check whether the request was made from an one of those IPs (here only the transmitting IP is checked since every request get duplicate pair of entries, one for the request, one for the response and all the other attributes are shared among that pair). If it is found, the parser also tries to identify the receiving IP address. If the receiving IP also matches, the request received and the counter for that particular service will increase. Then the parser moves on to record the number of bytes sent and received and the time taken to complete the request under identified service. Finally, these data points are exposed via an HTTP server so that the Prometheus scraper can read it and store it in the database so that it can be consumed by Sherlock.

```
def ipv4_request_event(self, cpu, data, size):
    event = self.b['ipv4_events'].event(data)
    # Decode data
    event = {
        ...
        ...
    }

    # Write to prometheus
    if event['LADDR'] in config_watcher.config:
        pod = config_watcher.config[event['LADDR']]
        if event['RADDR'] in config_watcher.config:
            rpod = config_watcher.config[event['RADDR']
            if not rpod['isService']:
                return
            request_received.labels(rpod['namespace'], rpod['serviceName'], rpod['name']).inc()
            request_exchanges.labels(pod['serviceName'], rpod['serviceName']).inc()

        ms.labels(pod['namespace'], pod['serviceName'], pod['name']).observe(event['MS'] / 1000000)
        tx_kb.labels(pod['namespace'], pod['serviceName'], pod['name']).observe(event['TX_KB'])
        rx_kb.labels(pod['namespace'], pod['serviceName'], pod['name']).observe(event['RX_KB'])
        request_sent.labels(pod['namespace'], pod['serviceName'], pod['name']).inc()
```

Figure 7.5: Code used to enrich TCP event data (self-composed)

7.3.3 AI-engine (Sherlock)

Sherlock is the AI engine that predicts anomaly scores for each service, which, in turn, is used by Operator to figure out possible root causes of a particular problem. From a high level, this works by polling service telemetry for a predetermined number of time steps and running it through a convolutional autoencoder which tries to reconstruct the input data sequence. The difference between the input sequence and the output sequence is called the reconstruction error, and this will be used as the anomaly score for this specific service. Even though this process seems straight forward, a number of prepocessing steps has to be taken in order to make it easier on the model to converge on the learning goal.

Since the collected metric data has different units, each feature of the dataset has a different range. This makes the training process very inefficient hence the model has to learn the concept of scales and unit first, and the backpropagation algorithm works best when the output values of the network are between 0-1 (Sola and Sevilla, 1997). So, to normalise this dataset, a slightly modified version of the Min-Max normalisation equation was used. This was done due to the fact that in most typical conditions metric values fluctuate between a fixed and limited range. If the Min-Max Normalisation Function is applied as if, the model may be hypersensitive to the slightest fluctuation. So adding this padding on both high and low ends acts as an attention mechanism that helps the model to focus on large variations rather than smaller ones.

```
def normalize(df):
    if df.name == "serviceName":
        return df
    else
        return (df-(df.min()))/((df.max()*1.3)-(df.min() * 1.3))
```

Figure 7.6: Data normalization function (self-composed)

feature	value
timestamp	2022-02-18 17:39:49
serviceName	service-1
acknowledged_bytes_per_minute	13405.6981119284
cpu_usage	307586440
high_syn_backlog_per_minute	158.5333333333
memory_usage	98295808
requests_duration_per_minute	0.0767269384
requests_received_per_minute	18.6440677966
requests_sent_per_minute	26.6101694915
syn_backlog_per_minute	69
transmitted_bytes_per_minute	14914.9181383682

(a) Before Normalization

feature	value
timestamp	2022-02-18 17:39:49
serviceName	service-1
acknowledged_bytes_per_minute	0.6018715135822673
cpu_usage	0.7692307692307694
high_syn_backlog_per_minute	0.3313198731853069
memory_usage	0.7692307692307689
requests_duration_per_minute	0.14394416054852854
requests_received_per_minute	0.6272133590680543
requests_sent_per_minute	0.6044198696736983
syn_backlog_per_minute	0.5539113198839827
transmitted_bytes_per_minute	0.5820578712565962

(b) After Normalization

Figure 7.7: Comparsion of a data point before and after the data normalization (self-composed)

During the requirement engineering process it was found out even though RNN tends to perform better with time-series data, the convolutional autoencoders are very efficient at detecting anomalies from time-series data. So after the normalization step metric data is encoded into an image-like structure that can be inputted into a convolutional autoencoder.

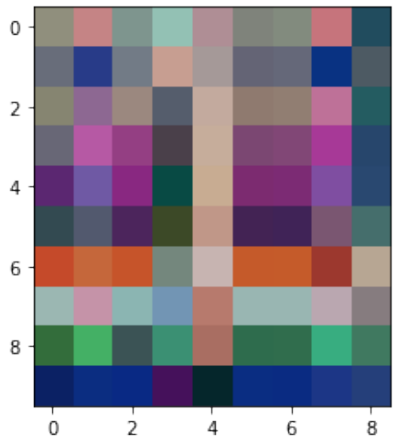


Figure 7.8: Visualization of encoded time series (self-composed)

7.4 Chapter Summary

In this chapter, the authors share their experiences and findings while implementing the proposed system. At the start, the author broke down the entire tech stack and explained all the tools and technologies used to build this project. Finally, the chapter was concluded with an explanation of the inner workings of the three core components and how they network to produce the result.

8 Testing

8.1 Chapter Overview

The objective of this chapter is to see the implementation of the project with the alignment of the design principle the author has mentioned above. To do that the author will first come up with a procedure to test the system. After that, the author will apply that procedure to the current implementation and present its results.

8.2 Objectives and Goals of Testing

The testing process in SDLC is to see whether all the key components are working as expected in order to call the project a success. Following are the key points that will be tested to see if the system is performing as planned.

- Validate all three components of the system work both individually and cooperatively to achieve the end goal.
- Understand how the system works on edge conditions.
- Identify if there are any potential Performance bottlenecks.
- Uncover any bugs or missing features that were left out during the implementation.
- Ensure all the must-haves and should-haves of the systems are implemented correctly.

8.3 Testing Criteria

A criterion is defined for testing the system in two ways with the goal of closing the gap between the expected and implemented systems.
The two methods of testing are as follows:

- Functional Quality - This focuses on the system's development characteristics and required technical aspects to see if it fits the given design based on functional requirements.
- Structural Quality - This tests the system's nonfunctional requirements while ensuring that it meets the performance requirements.

8.4 Testing Setup

Since the system is designed to integrate with an existing microservice system, the author needed a way to simulate the conditions of such a system with an emphasis on predictability and reliability. To achieve that, the author himself created a tool called MicroSim. This leverages

the extensibility of Kubernetes to provide a simplified API to the user to deploy a predictable microservice system.

8.5 Functional Testing

A series of black-box tests was performed on the system to find whether all functional requirements have been met and functioning as expected.

#	Description	System Input	Expected Outcome	Actual Outcome	Status
1	Deploy the Operator	Apply Lazy Koala manifests	Provision all the resources	Provision all the resources	Pass
2	Remove the Operator	Delete Lazy Koala manifests	Remove all the provisioned resources	Remove all the provisioned resources	Pass
3	Track a new service	Deploy a new Inspector resource	Update both Gazer and Sherlock configs	Update both Gazer and Sherlock configs	Pass
4	See the list of deployed Inspector resources	Open the settings view	List of deployments with tracked services highlighted	List of deployments with tracked services highlighted	Pass
5	Schedule Gazer instances in every node	Deploy Operator	A instance of Gazer get deployed on every node	A instance of Gazer get deployed on every node	Pass
6	Scrape “inet_sock_set_state” events from node kernel	Deploy a instance of Gazer on node	Export “inet_sock_set_state” event data of interested IPs to Prometheus.	Export “inet_sock_set_state” event data of interested IPs to prometheus.	Pass
7	Scrape “sk_ack_backlog” events from node kernel	Deploy an instance of Gazer on node	Export “sk_ack_backlog” event data of interested IPs to prometheus.	Export “sk_ack_backlog” event data of interested IPs to prometheus.	Pass

8	Poll Kube API for CPU and memory usage data	Deploy an instance of Gazer on node	Export CPU and memory usage data of interested services to prometheus	Export CPU and memory usage data of interested services to prometheus	Pass
9	React to IP changes of interested services	Deploy Operator	When a pod of tracked deployment get rescheduled, gazer config get updated to reflect that change	When a pod of tracked deployment get rescheduled, gazer config get updated to reflect that change	Pass
10	Real time config update for Gazer	Operator updates the Gazer config	Gazer hot reloads the config and reflect on changes	Gazer hot reloads the config and reflect on changes	Pass
11	Real time config update for Sherlock	Operator updates the Sherlock config	Sherlock hot reloads the config and reflect on changes	Sherlock hot reloads the config and reflect on changes	Pass
12	Regularly calculate anomaly scores	Deploy a new Inspector resource	Aggregate metric data and use the specified model to calculate the anomaly score	Aggregate metric data and use the specified model to calculate the anomaly score	Pass
13	Visualization the service mesh to understand the root course	Open the dashboard	Query prometheus for "request_exchange" and derive a graph from it, along with service health	Query prometheus for "request_exchange" and derive a graph from it, along with service health	Pass
14	Clean up orphaned resources	Delete an existing Inspector resource	Destroy all the child resources of the parent resource	Destroy all the child resources of the parent resource	Pass

Table 8.1: Results of Functional Testing (self-composed)

$$\text{Functional test case pass rate} = \frac{\text{Number of passed test cases}}{\text{Total number of test cases}} = \frac{14}{14} = 100\%$$

8.6 Module and Integration Testing

Since this system was developed to scale and fit into the microservices eco-system. Different components of the systems are also developed as microservices. The following tables show how the interaction between all these components works.

8.6.1 Operator Integration Testing

Action	Integration	Expected Outcome	Actual Outcome	Status
Subscribe to Inspector Resources	kube-controller-manager	Get notifications about changes to all of the Inspector resources	Get notifications about changes to all of the Inspector resources	Pass
Handle new Inspector resource creation	kube-apiserver	Update Gazer and Sherlock configs to track the deployment	Update Gazer and Sherlock configs to track the deployment	Pass
Handle IP changes of pods	kube-controller-manager	When a pod get change, update Gazer config to track the new IP	When a pod get change, update Gazer config to track the new IP	Pass
Handle Inspector resource deletion	kube-apiserver	Update Gazer and Sherlock configs to untrack the deployment	Update Gazer and Sherlock configs to untrack the deployment	Pass

Table 8.2: Operator Integration Testing (self-composed)

$$\text{Operator integration test pass rate} = \frac{\text{Number of passed test cases}}{\text{Total number of test cases}} = \frac{4}{4} = 100\%$$

8.6.2 Gazer Integration Testing

Action	Integration	Expected Outcome	Actual Outcome	Status
Install Kernel headers	Linux Kernel	Download and install the relevant kernel headers	Download and install the relevant kernel headers	Pass

Submit Probs to the Kernel space	Linux Kernel	Submit the eBPF prob and create the communication buffer	Submit the eBPF prob and create the communication buffer	Pass
Subscribe to inotify API to get config updates	Linux Kernel	React to inotify updates and hot reload the configuration	React to inotify updates and hot reload the configuration	Pass
Config parsing	Operator	Parse the config given by Operator to extract information about interested ClusterIPs	Parse the config given by Operator to extract information about interested ClusterIPs	Pass
Extract service usage data	kube-apiserver	Pool k8s-api for CPU and memory usage	Pool k8s-api for CPU and memory usage	Pass
Persist data for later processing	Prometheus	Aggregate service all telemetry data and export to prometheus	Aggregate service all telemetry data and export to prometheus	Pass

Table 8.3: Gazer Integration Testing (self-composed)

$$\text{Gazer integration test pass rate} = \frac{\text{Number of passed test cases}}{\text{Total number of test cases}} = \frac{6}{6} = 100\%$$

8.6.3 Sherlock Integration Testing

Action	**Integration**	**Expected Outcome**	**Actual Outcome**	**Status**
Visualize the service graph	Prometheus	Query prometheus for request exchanges and create a visual representation of the service graph	Query prometheus for request exchanges and create a visual representation of the service graph	Pass
List monitored and unmonitored services	kube-apiserver	Create table view of monitored and unmonitored services along with an option to toggle the behavior	Create table view of monitored and unmonitored services along with an option to toggle the behavior	Pass

Toggle monitored and unmonitored status of a service	kube-apiserver	Translate user's input to Inspector resource specification	Translate user's input to Inspector resource specification	Pass

Table 8.4: Sherlock Integration Testing (self-composed)

$$\text{Sherlock integration test pass rate} = \frac{\text{Number of passed test cases}}{\text{Total number of test cases}} = \frac{3}{3} = 100\%$$

8.6.4 UI Integration Testing

Action	**Integration**	**Expected Outcome**	**Actual Outcome**	**Status**
Visualize the service graph	Prometheus	Query prometheus for request exchanges and create a visual representation of the service graph	Query prometheus for request exchanges and create a visual representation of the service graph	Pass
List monitored and unmonitored services	kube-apiserver	Create table view of monitored and unmonitored services along with an option to toggle the behavior	Create table view of monitored and unmonitored services along with an option to toggle the behavior	Pass
Toggle monitored and unmonitored status of a service	kube-apiserver	Translate user's input to Inspector resource specification	Translate user's input to Inspector resource specification	Pass

Table 8.5: UI Integration Testing (self-composed)

$$\text{UI integration test pass rate} = \frac{\text{Number of passed test cases}}{\text{Total number of test cases}} = \frac{3}{3} = 100\%$$

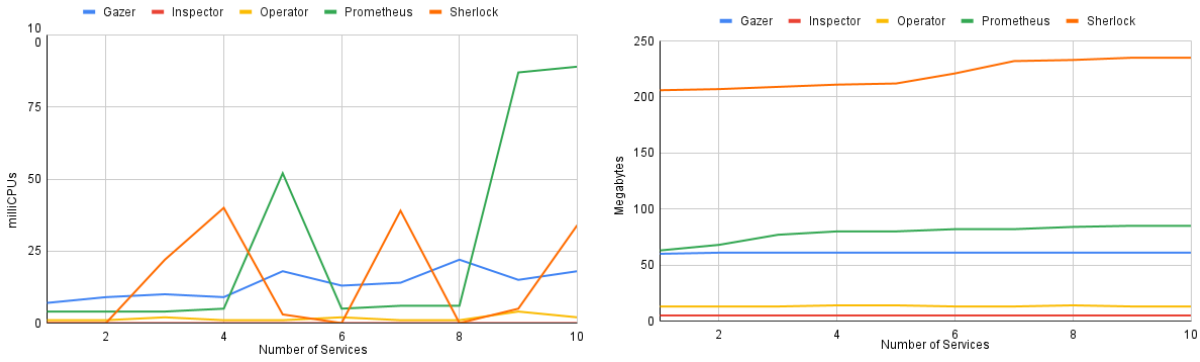
8.7 Non-Functional Testing

Non-functional testing is as important as functional testing since it is the basis for the quality of the product. It is important to ensure that the product is stable and reliable.

8.7.1 Performance Testing

One of the critical design goals of the system is to be able to use as few resources as possible while achieving all the functional requirements. To achieve this, the author used various techniques such as using system-level languages to write the core features and using modern coding practises such as coroutines to reduce resource waste.

The following diagram shows the relationship between the number of services that are being tracked versus the use of the system resource.



(a) CPU usage *(b) Memory usage*

Figure 8.1: Resource Usage vs Number of Services (self-composed)

8.7.2 Usability Testing

To increase the adaptability and usability of the system the author undertook a number of steps

- As shown in Section 8.6, all the components of the system are designed in a way that allows users to install individual components and still make use of them effectibly.
- Even though most of Kubernetes tooling revolved around terminals, for this system the author has created an intuitive UI so that users can interact without having to go though long documentations.
- The service graph is designed to be easily navigable and to allow users to easily find the relevant services.
- The entire application was packaged as a helm chart so the end-users can deploy or removed it to their cluster using a single command.

During the evaluation phase, the author demonstrated the end-to-end behaviour of the system and asked the evaluators to rate the usability of the system. The results of that are shown below.

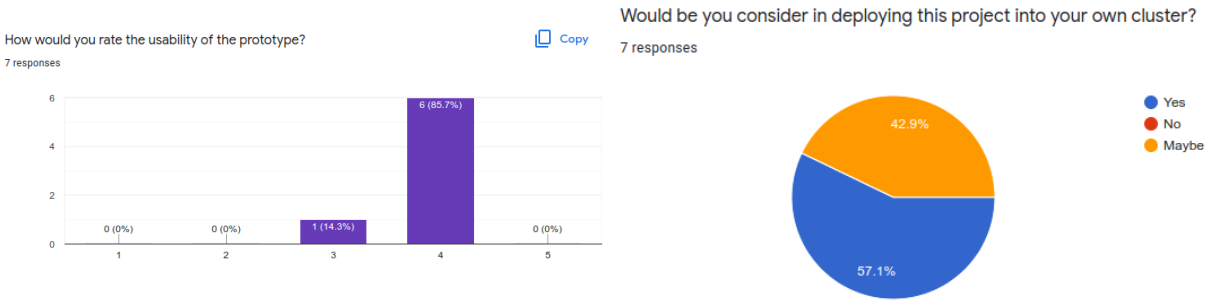


Figure 8.2: Usability of the system as rated by evaluator (self-composed)

8.7.3 Maintainability and Adaptability Testing

The main proposal of this project is to provide a framework for the research community to develop to push the limits of AIOps. So, from the very beginning, the system was consciously designed in a way that is easy to maintain and can be easily extended. Since quantifying the results of these effort is difficult, the author asked the evaluators number of questions to gauge the maintainability and adaptability of the system. The results of that are shown below.

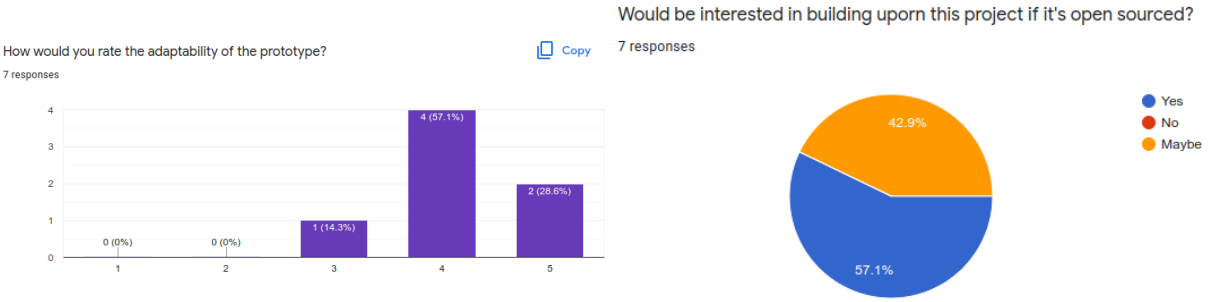
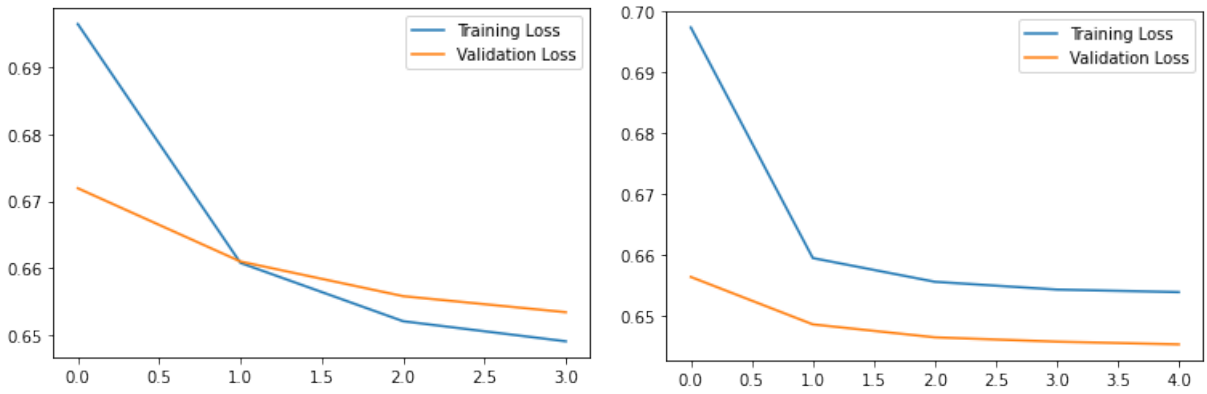


Figure 8.3: Maintainability and adaptability of the system as rated by evaluator (self-composed)

8.7.4 Generalisation Testing

Deep learning component of this system is based on the idea that at a fundamental level, all service acts the same. So, the author collected telemetry data from all services and trained a single model to understand the behaviour of the system. Once that based model is trained, it was fine tuned on the data collected from each service so the model get specialized to that particular service.



(a) Training loss of the base model

(b) Training loss of a specialized model

Figure 8.4: Generalizability of the Autoencoder (self-composed)

8.8 Limitations of the Testing Process

One of the main issues faced during the testing process is the lack of a proper benchmarking system and standard testing method to evaluate the performance of a framework like this. Therefore, the author has to come up with a method to evaluate this system. Another critical issue that could have strong implications was that the entire system was trained and tested on synthetic data, although the system shows strong indications it could adapt to real-world data, there may be more tuneing that needs to be done to work well with a production-level system.

8.9 Chapter Summary

This chapter forced the testing process of the proposed system. During this process, the author first explained the goals and criteria of the testing process. Then the testing process and the results of the tests were presented. Finally, the chapter concluded with limitations of the testing process and what could have been done better in the future.

9 Evaluation

9.1 Chapter Overview

The following chapter presents the evaluation process of this project and its findings. It starts by explaining the evaluation methodology and criteria for the project will be evaluated. The author then discusses their thoughts on the final product of this research. Then the thought process behind the selection evaluator will be explained. Finally, the chapter will conclude with feedback provided by the evaluators along with potential limitations in the evaluation.

9.2 Evaluation Methodology and Approach

Evaluation is vital to the success of any research project. It provides credibility to the claims of the research and its results. As mentioned in Section 3.2, the author has conducted a series of quantitative interviews with technical experts and domain experts, and has also conducted a qualitative survey to evaluate the system. The foci of these interviews and surveys are to evaluate the system's performance and impact on the real world by gathering data from an outside perspective. During the interviews, the author demonstrates the system's behaviour and different conditions, as well as the thinking process behind each of the features and its implementation.

9.3 Evaluation Criteria

As mentioned in section 9.2, the author conducted many interviews with experts in the field. Prior to each interview, the author got the consent of the evaluators to record the session. The following criteria were based on the themes extracted from the transcript of those interviews.

Criteria	**Purpose**
Overall impression about the project and research gap	Even Though cloud computing is a very matured field AIOps is still at its earliest stage. Therefore, validating the end for this research is critical to show the value addition done by this research.
Research scope and depth	In the author's opinion, the project had a very ambitious scope attached to it. Seeing an outside perspective to this will further validate the contributions made from the project.
Design and architecture	Since this project forces on microservices systems, the project itself was implemented as microservices, and getting a microservices specification just right is a complicated process.

Prototype implementation	As this meant to be a lightweight monitoring solution, it should have a minimal impact on the existing system. All the components were carefully constructed to minimize the resources wastage as possible. Getting an expert opinion on these design decisions will add more credibility to the author's claims.
Platform usability	As a plug and play system, it should be fairly easy for anyone with basic knowledge of Kubernetes to get it working within their system.
Contribution to the Domain	Validate the value addition done by the project and what it could mean for the future of AIOps.
Future work	Getting experts' opinions to see whether there are any implications on this project outside of being merely a research project. This will help to confirm the author's main goal of this system being a base for future researches.

Table 9.1: Evaluation Criteria (self-composed)

9.4 Self-Reflection

After almost nine months of hard work, the project was completed. During this time the author sets a lot of goals for themselves. The following table shows the author's thoughts on the progress of those goals and how well they were achieved and what could be improved upon.

Criteria	**Self-evaluation by the Author**
The overall impression about the project and research gap	The main bases for this project were the difficulties the author faced during their industrial placement. All existing service mesh-based visualisers require either a sidecar container or service-level telemetry reporting work. During the requirement elicitation process, it was found that the component Gazer alone will provide a great value in addition to the field.
Research scope and depth	During project proposal status, several industry experts were a bit concerned about the scope of this project and not being able to complete on time. Since the author was confident in their ability to bring such a system to life, they took it as a challenge that they could tackle.

Design and architecture	This project was designed to be lightweight, scalable, and decoupled as much as possible. So, from the programming language to the libraries that were used, were chosen very carefully. As shown in 8.7.1, the system is lightweight and all components of this are individually scalable.
Prototype implementation	This project contains 3 custom build microservices that can work individually or in collaboration to add value to the user. Operator built to extend Kubernetes' functionally. This is an extremely difficult process since the general consensus is Kubneates itself has a very steep learning curve (Anderson, 2021). Gazer is eBPF agent which directly interacts with the Linux Kernel. To achieve this, the author has to learn about kernel data structures and how they behave. Finally Sherlock, the AI engine is written in Rust to have the lowest memory and CPU footprint. To achieve this, the author had to implement some of the features such as matrix normalisation from the ground up, since Rust does not have a strong data science ecosystem like Python or R.
Platform usability	Usually Kubernetes based applications rely on "kubectl" utility to interact with. But this project offers a user-friendly web UI along with Kubectl integration so that end users can choose whatever they prefer to interact with the system. This approach allows power users like the author to interact with the system very efficiently while still allowing novices to access the same functionality using a GUI.
Contribution to the Domain	At the current stage, the project offers good value to SREs to understand the system and determine where the problems come from. Since this project is developed in a way which is easy to extend, the future researchers will have an easier time forcing on improving the prediction part without worrying about data gathering and processing.
Future work	One of the main key components that was left out of the project scope due to time constraints is giving the Operator the ability to constantly fine-tune the models so they can adapt to changes in service overtime. So, in the future, the author hopes to add this functionality as well.

Table 9.2: Self-evaluation by the author (self-composed)

9.5 Selection of the Evaluators

The author has decided to select a group of experts who have experience in both domains. Following are the credentials of the evaluators who were selected for this project.

ID	Name	Position	Affiliation	Reasons for Selection
EV1	Amila Maharachchi	VP of Engineering, Integration	WSo2	Experienced cloud engineer and possible end user
EV2	Sanjay Nadhavajhala	Engineer Manager, AI & Observability	SUSE	Maintainer of a similar scoped project called Opni
EV3	Matthias Rampke	Production Engineer	SoundCloud	Mentored the author since the beginning of the project
EV4	Lakmal Warusawithana	Senior Director - Cloud Architecture	WSo2	Experienced cloud engineer and have worked with eBPF
EV5	Shamal Perera	Senior Technical Lead	99x	Experienced cloud engineer and a lecturer
EV6	Jacob Payne	Senior Software Engineer	SUSE	Experienced Kubernetes developer and the author's mentor of Google Summer of Code
EV7	Nilesh Jayanandana	Solutions Architect	WSo2	Experienced cloud engineer who currently the Community Lead of Kubernetes Sri Lanka
EV8	Sameer Hamza	Senior Manager - Strategic Initiatives	Pearson	Experienced cloud engineer and a lecturer

Table 9.3: Selection of evaluators (self-composed)

9.6 Evaluation Results

9.6.1 Qualitative Result Analysis

The following table shows the overall impression of the interviewees after showing the demonstration of the system.

Code	Feedback
EV1	“Isala has done a tremendous amount of work to complete this project. It already covers a huge scope with multiple key technologies involved. It is definitely beyond the average projects we see within other undergraduates. Industry is definitely going to benefit from this concept and the prototype.”
EV2	“The project is very impressive!”
EV3	“Giving feedback and seeing progress has been a great experience!”
EV4	“Isala has done an excellent job. IMO, the scope of the project goes beyond the undergraduate project. Isala has used correct technologies in a correct way to solve the problem he tried to solve.”
EV5	“Overall it is a good project. There is a lot of work put behind to create the prototype, would like to see how it would perform by trying out models with different inputs.”
EV6	“This was a creative approach that I don’t know if I’ve seen anywhere else. The architecture is well designed and has appropriate divisions. Separating the metrics aggregation from the analysis was a good choice, as these components will scale differently.”
EV7	”Keep continuing working on the project. This is something really cool, and this could help a lot of startups to solve their problems because the key point in this is solving the service layer of related issues.”
EV8	“This looks really good, specifically for an undergrad project done in a limited time”

Table 9.4: Evaluator’s overall impression about the system (self-composed)

9.6.2 Quantitative Evaluation

Criteria	EV1	EV2	EV3	EV4	EV5	EV6	EV7	EV8
The overall impression about the project	**Does the proposed solution provide an excellent answer to the problem?** (1 - Strongly Disagree, 5 - Strongly Agree)							
	4	5	5	5	4	5	5	4
	The novelty of the project (1 - No novelty, 5 -Very high)							
	5	4	4	5	5	5	5	4
Research scope and depth	**Rate the complexity of the project** (1 - Not up to an undergraduate level, 5 - Could satisfy for a masters level)							
	4	3	5	5	5	5	5	4
	Rate the scope of the project (1 - Easy to achieve, 5 - Requires a lot of hard work)							
	5	4	5	5	5	5	5	5
Design and architecture	**The author showed a deep understanding of the subject matter** (1 - Strongly Disagree, 5 - Strongly Agree)							
	5	5	5	5	5	5	5	5
	The solution was well architectured to fit into microservices ecosystems (1 - Strongly Disagree, 5 - Strongly Agree)							
	5	5	5	5	4	4	5	4
Prototype implementation	**The project was well put together by following the best practices of both Dev and Ops** (1 - Strongly Disagree, 5 - Strongly Agree)							
	5	5	5	5	4	5	5	5
	The project was based on a cutting edge tech stack (1 - Strongly Disagree, 5 - Strongly Agree)							
	5	5	5	5	5	5	5	4
Platform usability	**How would you rate the usability of the prototype?** (1 - Poor, 5 - Excellent)							
	4	3	4	4	4	4	4	3
	How would you rate the adaptability of the prototype? (1 - Poor, 5 - Excellent)							
	4	3	5	4	4	5	4	4

Contribution to the Domain	**Impact of this project on the future of AIOps** (1 - No Impact at all, 5 - Very High)							
	5	4	5	5	4	5	5	4
Future work	**Would you consider deploying this project into your own cluster?** (Yes / Maybe / No)							
	Maybe	Yes	Yes	Maybe	Yes	Yes	Maybe	Yes
	Would you be interested in building upon this project if it's open-sourced? (Yes / Maybe / No)							
	Yes	Maybe	Yes	Maybe	Yes	Maybe	Yes	Yes

Table 9.5: Summary of the project evaluation (self-composed)

9.7 Limitations of Evaluation

Due to time constraints and lack of resources, the evaluation process of this project had to endure some limitations. The main limitation in this process is that, there is a heavily biased towards the technical experts when compared to domain experts. This was because the actual academics in the domain of cloud computing are hard to come by.

9.8 Evaluation on Functional Requirements

All the functional requirements were implemented and tested successfully. A detailed analysis of the results is available in the Appendix F.1.

9.9 Non Evaluation on Functional Requirements

All the non-functional requirements were met in the proposed solution and successfully tested. A detailed analysis of the results is available in the Appendix F.2.

9.10 Chapter Summary

This chapter discussed the final evaluation of the project by getting outside opinions on the system and analysing those data to see whether the original goals were met during the phase of this research.

10 Conclusion

10.1 Chapter Overview

The final chapter of this thesis presents a reflection on the entire project and its findings. Within this, the author will explain the project's goals and objectives along with their achievements. Skills and knowledge gains will be explained with the support of learning outcomes. Finally, the chapter will conclude with the limitations of the project and possible future enhancements.

10.2 Achievement of Research Aims & Objectives

10.2.1 Aim of the Project

The aim of this research is to design, develop and evaluate a low overhead Kubernetes framework to collect, store and process telemetry data using deep learning to help system operators detect anomalies earlier in order to reduce the MTTR when the system is experiencing an anomaly.

The aim of the project was successfully achieved by designing, developing, and evaluating a flexible framework which made with interchangeable components. This makes it easy for the data scientist to create reliable machine learning models to find the root causes of anomalies, which in turn help the system operators to reduce the MTTR.

10.2.2 Research Objectives

Research Objective	**Justification**	**Status**
Problem identification	A project with proper scope was identified to research on.	Completed
Literature review	A detailed literature survey was conducted on all related topics.	Completed
Developing an evaluation framework	MicroSim was developed and published.	Completed
Data gathering and analysis	eBPF agent Gazer and data preprocessor Sherlock was developed.	Completed
Developing encoding methods	Sherlock uses the RGB colour scheme to represent the telemetry data.	Completed
Developing the model	An autoencoder was developed to detect anomalies.	Completed

Testing and evaluation	Both functional and non-functional tests were carried out to verify the capabilities of the project. Along with third party evaluation.	Completed
Integration	Operator was built to bind up all the components together.	Completed

Table 10.1: Achievements of Research Objectives(self-composed)

10.3 Utilization of Knowledge from the Course

Module	**Description**
Programming Principles 01 and 02	Knowledge gained about software design concepts such as use case diagrams and flowcharts were reused to complete chapter 6 of this thesis.
Software Development Group Project	During this we were tasked with designing and developing a system from ground up. So, to achieve that, we were taught how to follow SDLC along with proper documentation.
Algorithms: Theory Design and Implementation	This module laid the foundation for us to help in terms of time and space complexity instead of solely relying on the results. The learnings from this module were taken to optimise both the CPU and memory footprint of the system.
Applied AI	Applied AI module was forced on applying concepts of artificial intelligence to real world problems. The learnings from this module were used when developing the Sherlock module.

Table 10.2: Utilization of knowledge gained from the course (self-composed)

10.4 Use of Existing Skills

Coming to this project, the author had some prior experiences with the area of the research. The skills reused from those experiences are listed below.

- The author had about four years of hands-on experience with Kubernetes due to their involvement with a startup.
- Machine learning is a key part of the project. Since their school days, the author had a

keen eye for the subject matter.

- During 2021's Google Summer of Code, the author worked on a project called Logging Pipeline Plumber for Rancher under the mentorship of Mr. Jacob Payne. The author learnt about the Kubernetes operator framework during that time.

10.5 New Skills Acquired

- **Research Skills** - During the course of this project, the author gained many hands-on ways of research and development along with how to present their findings.
- **eBPF** - To create the Gazer components, the author had to learn about a complex Linux kernel technology which will become heavily used with Kubernetes in coming years.
- **Rust Language** - Rust is a relatively new system-level language that is known for its resource efficiency and memory safety.

10.6 Achievement of Learning Outcomes

What was Learnt	**LOs**
During this project, the author learnt how to conduct a literature survey and critically analyse published work to understand the gaps between the published works.	LO1, LO3, LO4
Since this is an individual project, the author had to create a work schedule along with the goals they needed to achieve within the deadlines. This helped the author to learn to work autonomously with minimal supervision.	LO2, LO5, LO7
Since the beginning of the project, the author maintained best ethical practice by actively planning and mitigating SLEP issues mentioned in the chapter 5.2.	LO6
During this project, the author developed and properly documented the creation of a novel framework that will help to extend the field of AIOps.	LO7, LO8
At the end of the project, the author networked with industry experts to get their feedback and opinions about the developed system to support the author's claims.	LO8

Table 10.3: Achievement of Learning Outcomes (self-composed)

10.7 Problems and Challenges Faced

- **Learning Curve of eBPF** - As an optimisation step, the author decided to opt-in for using one of the Linux Kernel's subsystems called eBPF. This forced the author to learn some

of the inner workings of the Linux Kernel which is written in C.

- **Scope of the project** - Since this project touches both cloud computing and deep learning, there was a massive area to be explored.
- **Cost association with Kubernetes** - This project can not be deployed into the Kubernetes development environment since the Gazer requires direct access to a Linux Kernel. So the other had to use a production-grade Kubernetes cluster hosted in Google Cloud.
- **Economic Crisis** - With the end of COVID-19, Sri Lanka was faced with a massive economic crisis. Due to this, there were power cuts that lasted 10 hours at times. This caused the author to lose a lot of precious on working hours while sitting idle.

10.8 Deviations

The initial idea for this project was solely forced on the machine learning aspect to create a machine learning model that is highly specialized in service anomaly detection. But during prototyping, the author found out about the lack of infrastructure around AIOps. So, the scope of the project was broadened to include all the telemetry collection and processing steps too.

10.9 Limitations of the Research

- **Synthetic Data** - This system was both trained and tested using synthetic data. Even though there is a fair bit of randomness was injected to simulate the noisiness. The system may behave differently with actual production data.
- **Limited amount of telemetry** - Currently Gazer is only able to collect nine data points on a service. Increasing the number of collected metrics could lead to better results.
- **Limited to HTTP** - The system can only recognize HTTP-based communication between services. Communication protocols like WebSockets will require additional work.

10.10 Future Enhancements

- As mentioned in chapter 2 microservices tend to change over time. Extending Operator to automatically fine-tune the models would help overcome this problem.
- Gazer currently only works with low-level socket metrics. Adding support for more complex metrics, such as error rates returned from service, on the application layer could boost the accuracy of detections.
- With the current implementation Operator polling every deployment every minute to verify changes in pods IPs. Moving this to the pub/sub model will make the system react faster.

- Even though the model provided with Sherlock can detect anomalies with a fair bit of accuracy. There is a lot of optimisation that can be done to improve the accuracy of the model. For example, introducing LSTM cells could lead to better recall of past data.
- Supporting sharding in Sherlock will help with the deployment in larger environments so that the service could to be split across multiple VMs.

10.11 Contribution to the Body of Knowledge

10.11.1 Technical Contribution

- **MicroSim** - A tool to create a simulated distributed system within a Kubernetes Cluster. This tool also consists of a load generator which can create semi-random traffic patterns.
- **Lazy-Koala** - A Kubernetes native toolkit that provides a flexible API for collecting telemetry with zero instrumentation. This also helps to integrate any anomaly detection model into a system with minimal effort.

10.11.2 Domain Contribution

- **Data Encoding Technique** - This project introduced a novel data encoding technique that uses the RGB colour spectrum to represent telemetry data from different sampling windows. This helps both humans and machine learning models to understand the status of services.
- **Convolutional Neural Network** - To accompany the encoded data, a convolutional neural network was developed to detect anomalies. This model was trained on a large dataset of telemetry data collected from different services, which was built on a different framework. It helped to make the model very generalisable and easy to fine-tune per-service basis.

10.12 Concluding Remarks

In this thesis, the author proposed a collection of tools that can collect, store and process telemetry data from a distributed system to detect anomalies within the system. The system was demonstrated to several technical experts from both local and foreign organizations during the evaluation phase. They all shared their sentiments and stated that the research outcome has excellent potential if marketed correctly. Few of them were impressed with the author's accomplishments while working on this project and had sent offers to join their organizations. Finally the author hopes to open source the entire project under the MIT license so that both DevOps and data science communities can build on this to push the boundaries of AIOps.

References

Akcay, S., Atapour-Abarghouei, A. and Breckon, T. P. (2018), Ganomaly: Semi-supervised anomaly detection via adversarial training, *in* 'Asian conference on computer vision', Springer, pp. 622–637.

al dhuraibi, Y., Fawaz, P., Djarallah, N. and Merle, P. (2017), 'Elasticity in cloud computing: State of the art and research challenges', *IEEE Transactions on Services Computing* **PP**, 1–1.

Anderson, T. (2021), 'Google admits kubernetes container tech is so complex, it's had to roll out an autopilot feature to do it all for you • the register', `https://www.theregister.com/2021/02/25/google_kubernetes_autopilot/`. (Accessed on 09/14/2021).

Andy Honig, N. P. (2017), '7 ways we harden our kvm hypervisor at google cloud: security in plaintext — google cloud blog', `https://cloud.google.com/blog/products/gcp/7-ways-we-harden-our-kvm-hypervisor-at-google-cloud-security-in-plaintext`. (Accessed on 10/02/2021).

Batista, G. E., Prati, R. C. and Monard, M. C. (2004), 'A study of the behavior of several methods for balancing machine learning training data', *ACM SIGKDD explorations newsletter* **6**(1), 20–29.

Beyer, B., Jones, C., Petoff, J. and Murphy, N. (2016), *Site Reliability Engineering: How Google Runs Production Systems*, O'Reilly Media, Incorporated.
URL: *https://books.google.lk/books?id=81UrjwEACAAJ*

Chaczko, Z., Mahadevan, V., Aslanzadeh, S. and Mcdermid, C. (2011), Availability and load balancing in cloud computing, *in* 'International Conference on Computer and Software Modeling, Singapore', Vol. 14, IACSIT Press, pp. 134–140.

Chappell, E. (2021), 'What are the top 8 project management methodologies?', `https://clickup.com/blog/project-management-methodologies/#36-7-what-is-the-prince2-methodology`. (Accessed on 09/22/2021).

Chigurupati, A. and Lassar, N. (2017), Root cause analysis using artificial intelligence, *in* '2017 Annual reliability and maintainability symposium (RAMS)', IEEE, pp. 1–5.

CNCF (2020), 'Cloud native survey 2020: Containers in production jump 300% from our first survey', https://www.cncf.io/blog/2020/11/17/cloud-native-survey-2020-containers-in-production-jump-300-from-our-first-survey/ (Accessed on 02/02/2022).

Conventional Commits (n.d.), https://www.conventionalcommits.org/en/v1.0.0-beta.2/. (Accessed on 05/15/2022).

Dasgupta, D. and Majumdar, N. S. (2002), Anomaly detection in multidimensional data using negative selection algorithm, *in* 'Proceedings of the 2002 Congress on Evolutionary Computation. CEC'02 (Cat. No. 02TH8600)', Vol. 2, IEEE, pp. 1039–1044.

Di Francesco, P., Lago, P. and Malavolta, I. (2018), Migrating towards microservice architectures: an industrial survey, *in* '2018 IEEE International Conference on Software Architecture (ICSA)', IEEE, pp. 29–2909.

Donahue, J., Krähenbühl, P. and Darrell, T. (2016), 'Adversarial feature learning', *arXiv preprint arXiv:1605.09782* .

Du, Q., Xie, T. and He, Y. (2018), Anomaly detection and diagnosis for container-based microservices with performance monitoring, *in* 'International Conference on Algorithms and Architectures for Parallel Processing', Springer, pp. 560–572.

Dua, R., Raja, A. R. and Kakadia, D. (2014), Virtualization vs containerization to support paas, *in* '2014 IEEE International Conference on Cloud Engineering', IEEE, pp. 610–614.

Dudovskiy, J. (n.d.), 'Pragmatism research philosophy', https://research-methodology.net/research-philosophy/pragmatism-research-philosophy/. (Accessed on 09/08/2021).

Galov, N. (2021), 'Cloud adoption statistics - it's everywhere & everyone's using it in 2021!', https://hostingtribunal.com/blog/cloud-adoption-statistics/. (Accessed on 08/27/2021).

Geethika, D., Jayasinghe, M., Gunarathne, Y., Gamage, T. A., Jayathilaka, S., Ranathunga, S. and Perera, S. (2019), Anomaly detection in high-performance api gateways, *in* '2019 International Conference on High Performance Computing & Simulation (HPCS)', IEEE, pp. 995–1001.

Ghasedi Dizaji, K., Herandi, A., Deng, C., Cai, W. and Huang, H. (2017), Deep clustering via joint convolutional autoencoder embedding and relative entropy minimization, *in* 'Proceedings of the IEEE international conference on computer vision', pp. 5736–5745.

Gillis, A. S. (2019), 'What is sidecar proxy?', `https://searchitoperations.techtarget.com/definition/sidecar-proxy`. (Accessed on 10/02/2021).

Gondara, L. (2016), Medical image denoising using convolutional denoising autoencoders, *in* '2016 IEEE 16th international conference on data mining workshops (ICDMW)', IEEE, pp. 241–246.

Gonzalez, J. M. N., Jimenez, J. A., Lopez, J. C. D. et al. (2017), 'Root cause analysis of network failures using machine learning and summarization techniques', *IEEE Communications Magazine* **55**(9), 126–131.

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A. and Bengio, Y. (2014), 'Generative adversarial nets', *Advances in neural information processing systems* **27**.

Google Cloud Incident #20013 (2020), `https://status.cloud.google.com/incident/zall/20013`. (Accessed on 12/03/2021).

Hagemann, T. and Katsarou, K. (2020), A systematic review on anomaly detection for cloud computing environments, *in* '2020 3rd Artificial Intelligence and Cloud Computing Conference', pp. 83–96.

Hausenblas, M. and Schimanski, S. (2019), *Programming Kubernetes: Developing Cloud-Native Applications*, O'Reilly Media.
URL: *https://books.google.lk/books?id=7VKjDwAAQBAJ*

Hinton, G. E. and Salakhutdinov, R. R. (2006), 'Reducing the dimensionality of data with neural networks', *science* **313**(5786), 504–507.

IBM (2021), 'What is container orchestration?', `https://www.ibm.com/cloud/learn/container-orchestration`. (Accessed on 09/14/2021).

JetBrains (n.d.), 'What are the benefits of ci/cd?', `https://www.jetbrains.com/teamcity/ci-cd-guide/benefits-of-ci-cd/`. (Accessed on 08/26/2021).

Khononov, V. (2020), 'Untangling microservices or balancing complexity in distributed systems', https://blog.doit-intl.com/untangling-microservices-or-balancing-complexity-in-distributed-systems-7759987d4 (Accessed on 08/31/2021).

Khoshnevisan, F. and Fan, Z. (2019), 'Rsm-gan: A convolutional recurrent gan for anomaly detection in contaminated seasonal multivariate time series', *arXiv preprint arXiv:1911.07104* .

Kim, G., Behr, K. and Spafford, K. (2014), *The Phoenix Project: A Novel About IT, DevOps, and Helping Your Business Win*, IT Revolution Press.
URL: *https://books.google.lk/books?id=qaRODgAAQBAJ*

Kim, T., Suh, S. C., Kim, H., Kim, J. and Kim, J. (2018), An encoding technique for cnn-based network anomaly detection, *in* '2018 IEEE International Conference on Big Data (Big Data)', IEEE, pp. 2960–2965.

Kivity, A., Kamay, Y., Laor, D., Lublin, U. and Liguori, A. (2007), kvm: the linux virtual machine monitor, *in* 'Proceedings of the Linux symposium', Dttawa, Dntorio, Canada, pp. 225–230.

Kumarage, T., De Silva, N., Ranawaka, M., Kuruppu, C. and Ranathunga, S. (2018), Anomaly detection in industrial software systems-using variational autoencoders., *in* 'ICPRAM', pp. 440–447.

Kumarage, T., Ranathunga, S., Kuruppu, C., De Silva, N. and Ranawaka, M. (2019), Generative adversarial networks (gan) based anomaly detection in industrial software systems, *in* '2019 Moratuwa Engineering Research Conference (MERCon)', IEEE, pp. 43–48.

Li, W., Lemieux, Y., Gao, J., Zhao, Z. and Han, Y. (2019), Service mesh: Challenges, state of the art, and future research opportunities, *in* '2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)', IEEE, pp. 122–1225.

Liker, J. K. and Meier, D. (2006), *Toyota way fieldbook*, McGraw-Hill Education.

Linders, B. (2021), 'Artificial intelligence for it operations: an overview', https://www.infoq.com/news/2021/07/AI-IT-operations/. (Accessed on 12/04/2021).

Ma, M., Xu, J., Wang, Y., Chen, P., Zhang, Z. and Wang, P. (2020), Automap: Diagnose your microservice-based web applications automatically, *in* 'Proceedings of The Web Conference 2020', pp. 246–258.

Menezes, B. (2018), 'Watchdog: Auto-detect performance anomalies without setting alerts — datadog', https://www.datadoghq.com/blog/watchdog/. (Accessed on 08/29/2021).

Meng, Y., Zhang, S., Sun, Y., Zhang, R., Hu, Z., Zhang, Y., Jia, C., Wang, Z. and Pei, D. (2020), Localizing failure root causes in a microservice through causality inference, *in* '2020 IEEE/ACM 28th International Symposium on Quality of Service (IWQoS)', IEEE, pp. 1–10.

Mergen, M. F., Uhlig, V., Krieger, O. and Xenidis, J. (2006), 'Virtualization for high-performance computing', *ACM SIGOPS Operating Systems Review* **40**(2), 8–11.

Microsoft (2021), 'Microsoft 365 status on twitter: "we're investigating an issue for access to multiple m365 services. please visit the admin center post m0244568 for more information. we'll provide additional information here as it becomes available."', https://twitter.com/MSFT365Status/status/1371546946263916545. (Accessed on 03/02/2022).

Mike Loukides, S. S. (2020), 'Microservices adoption in 2020 – o'reilly', https://www.oreilly.com/radar/microservices-adoption-in-2020/. (Accessed on 09/22/2021).

Mishra, S. (2017), 'Unsupervised learning and data clustering', https://towardsdatascience.com/unsupervised-learning-and-data-clustering-eeecb78b422a. (Accessed on 10/08/2021).

Molnar, I. (2015), 'Lkml: Ingo molnar: [git pull] perf updates for v4.1', https://lkml.org/lkml/2015/4/14/232. (Accessed on 10/02/2021).

Morgan, W. (2021), 'Benchmarking linkerd and istio', https://linkerd.io/2021/05/27/linkerd-vs-istio-benchmarks/. (Accessed on 10/02/2021).

Munim, Z. H. (2019), 'Philosophy of science — four major paradigms', https://www.youtube.com/watch?v=n8B50HJrAv0. (Accessed on 09/08/2021).

Nguyen, H., Shen, Z., Tan, Y. and Gu, X. (2013), Fchain: Toward black-box online fault localization for cloud systems, *in* '2013 IEEE 33rd International Conference on Distributed Computing Systems', IEEE, pp. 21–30.

Nguyen, H., Tan, Y. and Gu, X. (2011), Pal: ¡i¿p¡/i¿ropagation-aware ¡i¿a¡/i¿nomaly ¡i¿l¡/i¿ocalization for cloud hosted distributed applications, *in* 'Managing Large-Scale Systems via the Analysis of System Logs and the Application of Machine Learning Techniques', SLAML '11, Association for Computing Machinery, New York, NY, USA.
URL: *https://doi.org/10.1145/2038633.2038634*

Novak, A. (2016), 'Going to market faster: Most companies are deploying code weekly, daily, or hourly', https://newrelic.com/blog/best-practices/data-culture-survey-results-faster-deployment. (Accessed on 09/04/2021).

Oord, A. v. d., Dieleman, S., Zen, H., Simonyan, K., Vinyals, O., Graves, A., Kalchbrenner, N., Senior, A. and Kavukcuoglu, K. (2016), 'Wavenet: A generative model for raw audio', *arXiv preprint arXiv:1609.03499* .

OpenAI (2018), 'Openai five', https://blog.openai.com/openai-five/.

Prabodha, L., Vithanage, W., Ranaweera, L., Dissanayake, D. and Ranathunga, S. (2017), Monitoring health of large scale software systems using drift detection techniques, *in* 'Conference on Complex, Intelligent, and Software Intensive Systems', Springer, pp. 152–163.

RedHat (2021), 'Kubernetes adoption, security, and market trends report 2021', https://www.redhat.com/en/resources/kubernetes-adoption-security-market-trends-2021-overview. (Accessed on 11/20/2021).

RedHat (n.d.), 'Understanding cloud-native apps', https://www.redhat.com/cloud-native-apps. (Accessed on 08/26/2021).

Rimol, M. (2021), 'Gartner says worldwide iaas public cloud services market grew 40.7% in 2020', https://www.gartner.com/en/newsroom/press-releases/2021-06-28-gartner-says-worldwide-iaas-public-cloud-services-market-grew-40-7-per

Samir, A. and Pahl, C. (2019), Dla: Detecting and localizing anomalies in containerized microservice architectures using markov models, *in* '2019 7th International Conference on Future Internet of Things and Cloud (FiCloud)', IEEE, pp. 205–213.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M. et al. (2016), 'Mastering the game of go with deep neural networks and tree search', *nature* **529**(7587), 484–489.

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A. et al. (2017), 'Mastering the game of go without human knowledge', *nature* **550**(7676), 354–359.

Sola, J. and Sevilla, J. (1997), 'Importance of input data normalization for the application of neural networks to complex industrial problems', *IEEE Transactions on nuclear science* **44**(3), 1464–1468.

Soldani, J. and Brogi, A. (2021), 'Anomaly detection and failure root cause analysis in (micro) service-based cloud applications: A survey', *arXiv preprint arXiv:2105.12378* .

Spoonhower, D. (2018), 'Lessons from the birth of microservices at google', `https://dzone.com/articles/lessons-from-the-birth-of-microservices-at-google`. (Accessed on 09/22/2021).

The Kubebuilder Book (n.d.), `https://book.kubebuilder.io`. (Accessed on 02/03/2022).

Toka, L., Dobreff, G., Haja, D. and Szalay, M. (2021), Predicting cloud-native application failures based on monitoring data of cloud infrastructure, *in* '2021 IFIP/IEEE International Symposium on Integrated Network Management (IM)', IEEE, pp. 842–847.

Wang, L., Zhao, N., Chen, J., Li, P., Zhang, W. and Sui, K. (2020), Root-cause metric location for microservice systems via log anomaly detection, *in* '2020 IEEE International Conference on Web Services (ICWS)', IEEE, pp. 142–150.

Wang, Y., Yao, Q., Kwok, J. T. and Ni, L. M. (2020), 'Generalizing from a few examples: A survey on few-shot learning', *ACM Computing Surveys (CSUR)* **53**(3), 1–34.

WeaveWorks (2020), 'Docker vs virtual machines (vms) : A practical guide to docker containers and vms', `https://www.weave.works/blog/a-practical-guide-to-choosing-between-docker-containers-and-vms`. (Accessed on 11/20/2021).

What is eBPF? (n.d.), `https://ebpf.io/what-is-ebpf`. (Accessed on 10/07/2021).

What is virtualization? (n.d.), https://www.redhat.com/en/topics/virtualization/what-is-virtualization#history-of-virtualization. (Accessed on 10/02/2021).

Wiesen, G. (n.d.), 'What is a system monitor?', https://www.easytechjunkie.com/what-is-a-system-monitor.htm. (Accessed on 10/02/2021).

Wu, L., Tordsson, J., Elmroth, E. and Kao, O. (2020), Microrca: Root cause localization of performance issues in microservices, *in* 'NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium', IEEE, pp. 1–9.

Zhang, C., Song, D., Chen, Y., Feng, X., Lumezanu, C., Cheng, W., Ni, J., Zong, B., Chen, H. and Chawla, N. V. (2019), A deep neural network for unsupervised anomaly detection and diagnosis in multivariate time series data, *in* 'Proceedings of the AAAI Conference on Artificial Intelligence', Vol. 33, pp. 1409–1416.

Zhitnitsky, A. (2019), '5 ways you're probably messing up your microservices — overops', https://www.overops.com/blog/5-ways-to-not-f-up-your-microservices-in-production/. (Accessed on 08/26/2021).

Appendix A: Concept Map

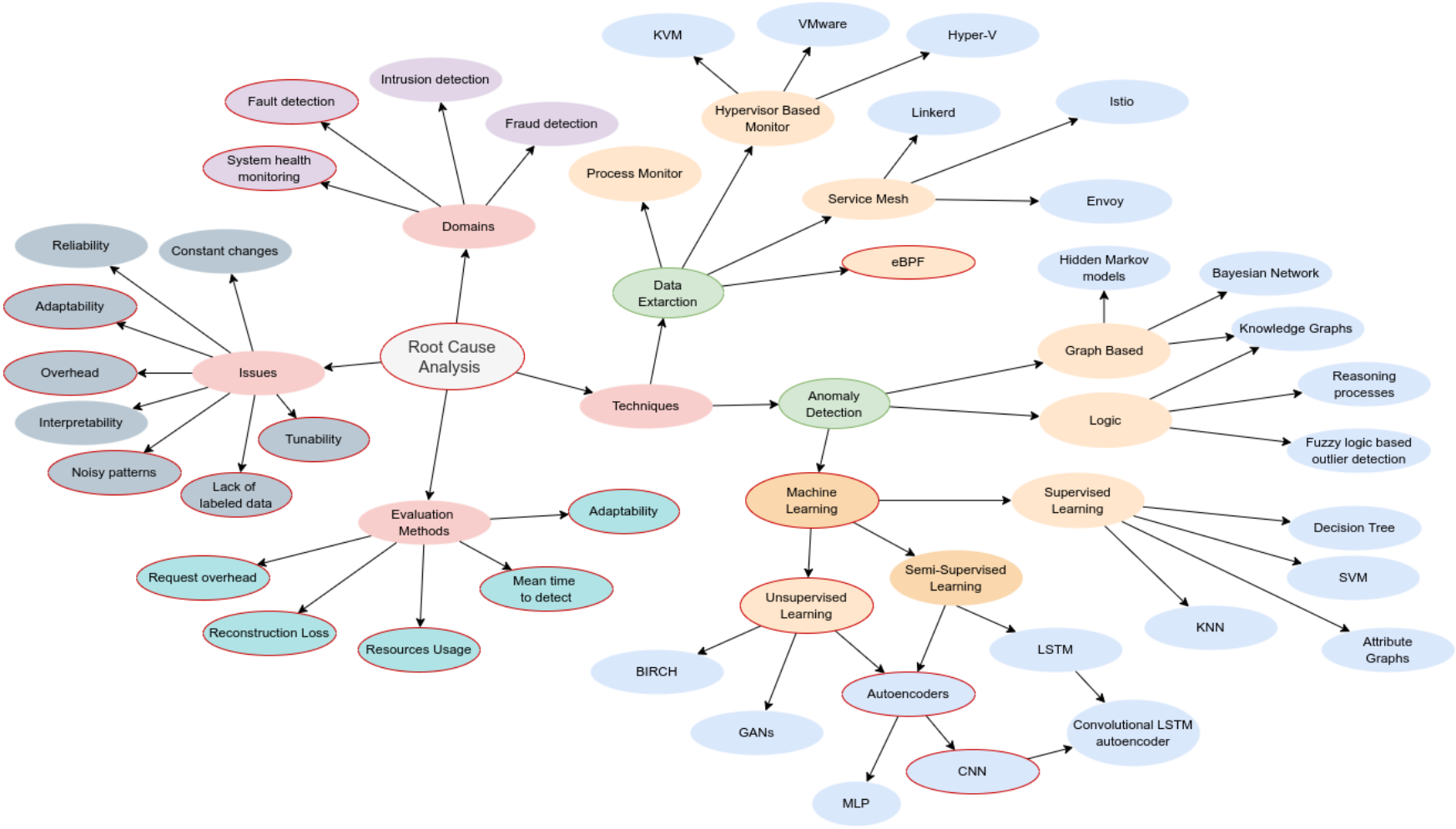


Figure A.1: Concept map (self-composed)

Appendix B: Gantt Chart

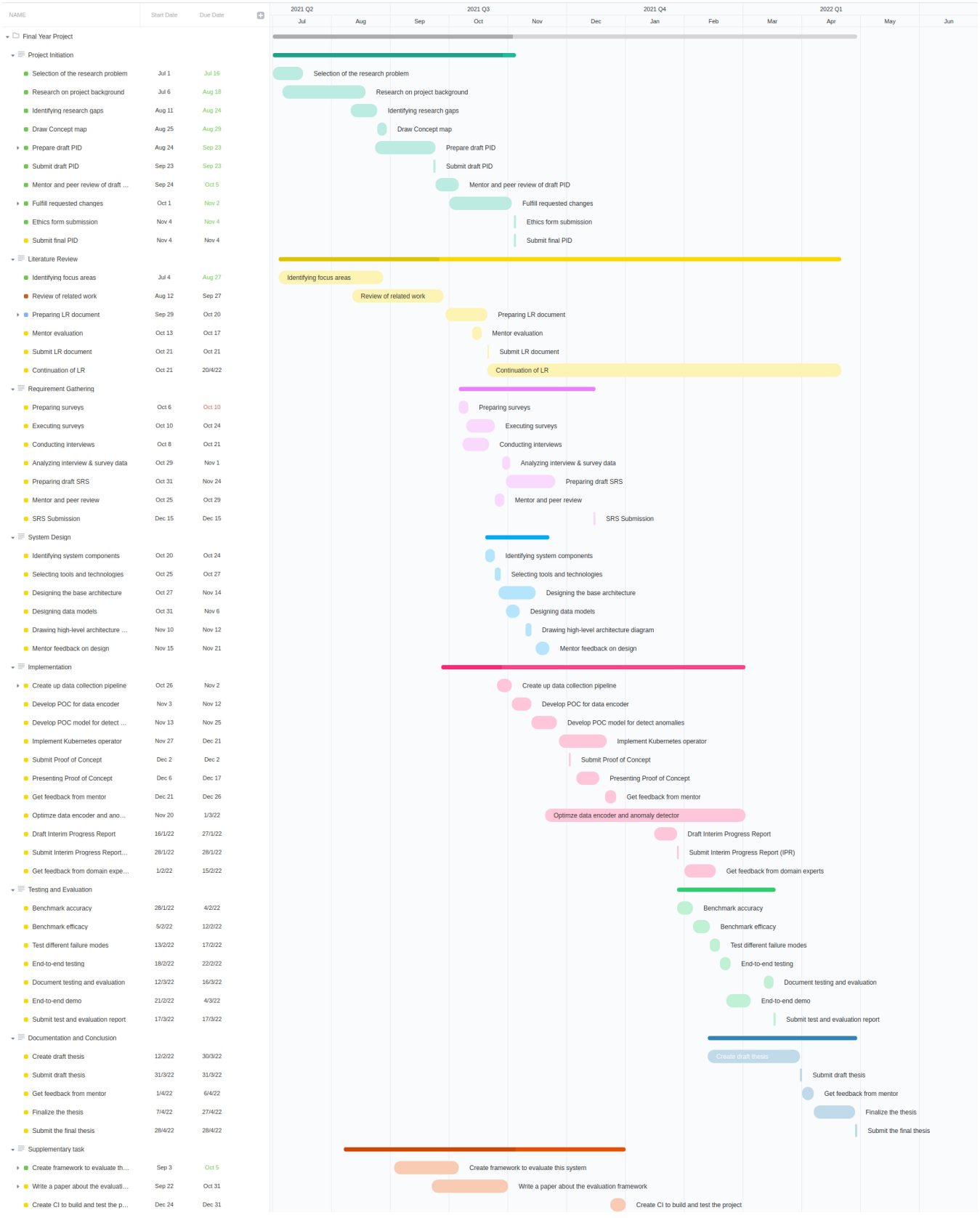


Figure B.1: Defined gantt chart for the project (self composed)

Appendix C: Use Case Descriptions

Use Case ID	**UC-01**
Use Case Name	Deploy Lazy Koala
Description	Install Operator to a Kubernetes cluster
Participating actors	Reliability Engineer
Preconditions	• A Kubernetes cluster running. • kubectl installed and configured to talk to the cluster. • Helm CLI installed.
Extended use cases	N/A
Included use cases	N/A
Main flow	1. Add Helm remote. 2. Run helm install command. 3. Kube API acknowledges the changes. 4. Display content of Notes.txt
Alternative flows	1. Apply Kubernetes Manifest found in the code repository. 2. Kube API acknowledges the changes.
Exceptional flows	**E1**: A Operator couldn't achieve desired state 1. The Operator retries to achieve the desired state with an exponential backoff
Postconditions	• Operator deployed on the cluster. • Instance of Gazer deployed on every node. • New permission rules are registered with Kube API.

Use Case ID	**UC-02**
Use Case Name	Update Configuration
Description	Add or Remove a service from a monitored list.
Participating actors	Reliability Engineer
Preconditions	• kubectl installed and configured to talk to a Kubernetes cluster. • The Kubernetes cluster has a Operator deployed. • Established port forwarding connection with Operator.
Extended use cases	N/A

Included use cases	N/A
Main flow	1. Visit the forwarded port on the local machine. 2. Open the “Services” tab. 3. Click Attach Inspector. 4. Select the namespace and the service. 5. Click Attach. 6. Status update sent to kube API.
Alternative flows	1. Visit the forwarded port on the local machine. 2. Open the “Services” tab. 3. Scroll to the relevant record. 4. Press the delete button next to the name. 5. Confirm the action. 6. Status update sent to kube API.
Exceptional flows	**E1**: Kube API not available 1. Show an error to the user asking to retry in a bit.
Postconditions	• A new Inspector resource is attached to the service.

Use Case ID	**UC-03**
Use Case Name	Purge Lazy Koala
Description	Remove Lazy Koala from a Kubernetes cluster.
Participating actors	Reliability Engineer
Preconditions	• kubectl installed and configured to talk to a Kubernetes cluster. • The Kubernetes cluster has a Operator deployed.
Extended use cases	N/A
Included use cases	N/A
Main flow	1. Find the helm release name. 2. Run helm uninstall [release name].
Alternative flows	1. Locate Kubernetes Manifest found in the code repository. 2. Run kubectl delete -f [manifest-file]
Exceptional flows	N/A

Postconditions	• All the resources provisioned by Lazy Koala including the Operator itself get removed from the cluster.

Use Case ID	**UC-07**
Use Case Name	Extract telemetry
Description	Every 5 second Gazer will scrape the metric server
Participating actors	System Timer
Preconditions	• Gazer is deployed to the cluster
Extended use cases	N/A
Included use cases	Write to the database
Main flow	1. poll_kube_api function get invoked. 2. Gazer looks at the config file and finds out the service it's responsible for. 3. Query metric server for each of the service names. 4. Store it in local memory.
Alternative flows	N/A
Exceptional flows	**E1**: metric server returns non 200 status code. 1. Retry in the next iteration.
Postconditions	• Updated local memory with recent telemetry data.

Use Case ID	**UC-09**
Use Case Name	Check for Anomalies
Description	Check for Anomalies in each of the service
Participating actors	System Timer
Preconditions	• An Instance of Sherlock is deployed.
Extended use cases	N/A
Included use cases	Write to the database

Main flow	1. check_anomlies function invoked. 2. Query the database for telemetry for about the last 5 minutes. 3. Do a forward pass on the model. 4. Calculate the reconstruction loss. 5. Store it in local memory.
Alternative flows	N/A
Exceptional flows	**E1**: Database is unreachable 1. Retry in the next iteration.
Postconditions	• Updated local memory with current reconstruction loss.

Use Case ID	**UC-11**
Use Case Name	Reconcile on modified resources
Description	Whenever a resource owned by the Operator gets modified, kubelet invokes the reconciliation loop on the Operator.
Participating actors	Kubelet
Preconditions	• Operator is deployed
Extended use cases	Read the cluster state
Included use cases	N/A
Main flow	1. Resources get modified. 2. Kubelet invokes a reconciliation loop on the Operator. 3. Check if the change is interesting. 4. Update children resources accordingly.
Alternative flows	1. Resources get modified. 2. Kubelet invokes a reconciliation loop on the Operator. 3. Check if the change is interesting. 4. Stop execution.
Exceptional flows	**E1**: Error while reconciling 1. Retry with exponential backoff.
Postconditions	• Cluster in the new desired state.

Appendix D: Proof of Concept Results

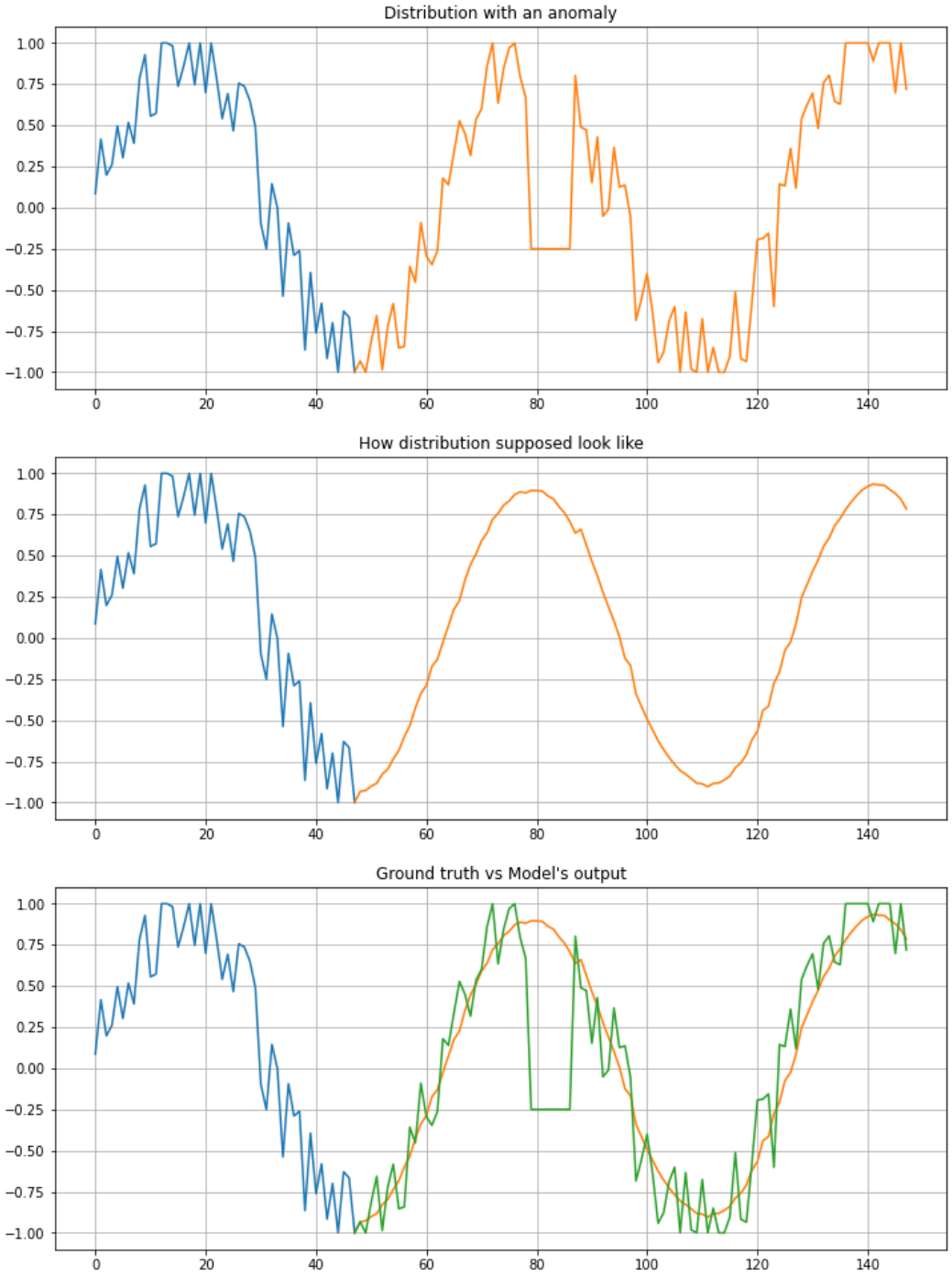


Figure D.1: Proof of concept results (self-composed)

Appendix E: Prometheus Dashboard

Figure E.1: Prometheus dashboard with collected data (self-composed)

Appendix F: Evaluations

F.1 Evaluation of the functional requirements

ID	Requirement and Description	Priority Level	Status
FR01	Users should be able to deploy the Operator to an existing Kubernetes cluster.	Must have	Implemented
FR02	Users should be able to remove the Operator completely from the cluster.	Should have	Implemented
FR03	Users should be able to specify which services need to be monitored.	Must have	Implemented
FR04	Users should be able to see the services monitored by Operator	Could have	Implemented
FR05	Operator should deploy an instance of Gazer to every node in the cluster.	Must have	Implemented
FR06	Gazer should intersect all "inet_sock_set_state" kernel calls and export the relevant data to Prometheus.	Must have	Implemented
FR07	Gazer periodically polls the size of "sk_ack_backlog" for interested ports to export the relevant data to Prometheus.	Should have	Implemented
FR08	Gazer should poll the Kubernetes metric server periodically and export the relevant data to Prometheus.	Must have	Implemented
FR09	Operator should periodically check for changes in monitored services and update the Gazer ConfigMap.	Should have	Implemented
FR10	Gazer should react to config updates in realtime.	Could have	Implemented
FR11	Sherlock should provision the list of models given by Operator	Must have	Implemented
FR12	Sherlock should periodically calculate the anomaly score for each of the monitored services and export it to Prometheus.	Must have	Implemented

FR13	Operator should have a Web UI to visualize the service topology.	Should have	Implemented
FR14	Operator should add a finalizer for each of the provisioned resources.	Should have	Implemented
FR15	Operator should periodically fine-tune models.	Will not have	Not Implemented

Table F.1: Evaluation of the functional requirements

F.2 Evaluation of the non-functional requirements

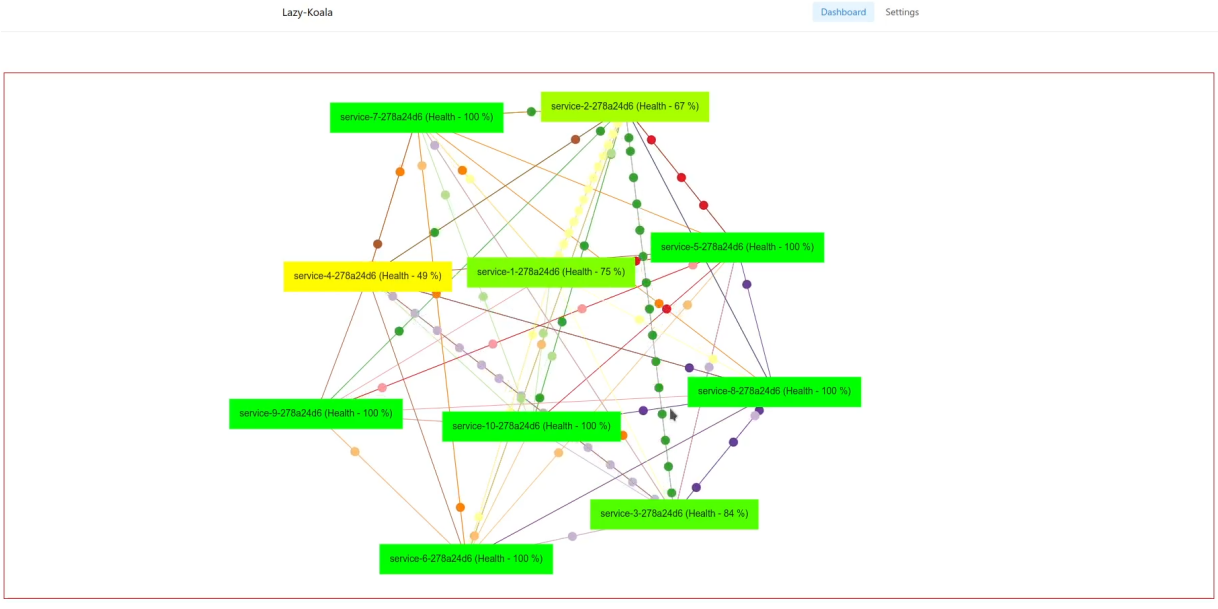
ID	**Description**	**Priority Level**	**Status**
NFR1	Operator should follow Principle of Least Privilege when accessing Kubernetes APIs.	Must have	Implemented
NFR2	Systems should have fragmented architecture so each component can be individually scaled in order to save resources.	Must have	Implemented
NFR3	Each component should work individually such that users can install parts of the system they are interested in.	Could have	Implemented
NFR4	Gazer should be limited for using only 100 mCPUs and 80MB of memory.	Should have	Implemented
NFR5	Sherlock should be limited to using only 100 mCPUs and 100MB of memory.	Could have	Implemented
NFR6	Operator should be packaged as a Helm Chart for ease of use.	Must have	Implemented
NFR7	Reconstruction error of Sherlock should be under 0.1%	Could have	Implemented
NFR8	Operator's reconciliation loop should use exponential backoff technique when there is an error while reconciling for a config change.	Must have	Implemented

NFR9	Follow Coding best practices and rely on linters for code formatting.	Could have	Implemented
NFR10	The project should be backed by an automated CI/CD tool to test and build each component with each release.	Could have	Implemented

Table F.2: Evaluation of the non-functional requirements

Appendix G: User Interface

(a) Service graph under normal conditions



(b) Service graph while experiencing an anomaly

Figure G.1: Main dashboard of the system (self-composed)

Appendix H: Installation of Lazy-Koala

```
 /mnt/s/P/II/FY/lazy-koala/charts  feat/helm
❯ helm install lazy-koala --generate-name -n lazy-koala --create-namespace
NAME: lazy-koala-1652462705
LAST DEPLOYED: Fri May 13 22:55:06 2022
NAMESPACE: lazy-koala
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
Lazy Koala was successfully installed the lazy-koala namespace

To access the main dashboard,
  1. Run
         $ kubectl port-forward svc/inspector 8090:80 -n lazy-koala
  2. Open http://localhost:8090 in your browser

Under the settings tab you will be able to find out list of service that's running inside the cluster.
From there you can mark them to be monitored by the gazer agent.

Finally, Once you navigated to the main dashboard, you can see the list of services that are being monitored
and how they interact with each other.

To Uninstall the Lazy Koala, Simply run
  $ helm uninstall lazy-koala-1652462705 -n lazy-koala
```

Figure H.1: Helm chart installation output (self-composed)

```
 /mnt/s/P/II/FY/lazy-koala/charts  feat/helm
❯ helm uninstall lazy-koala-1652462705 -n lazy-koala
release "lazy-koala-1652462705" uninstalled

 /mnt/s/P/II/FY/lazy-koala/charts  feat/helm
❯ kubectl get all -n lazy-koala
No resources found in lazy-koala namespace.
```

Figure H.2: Uninstallation of operator (self-composed)

Appendix I: Use of Best Practices

I.1 Project Structure

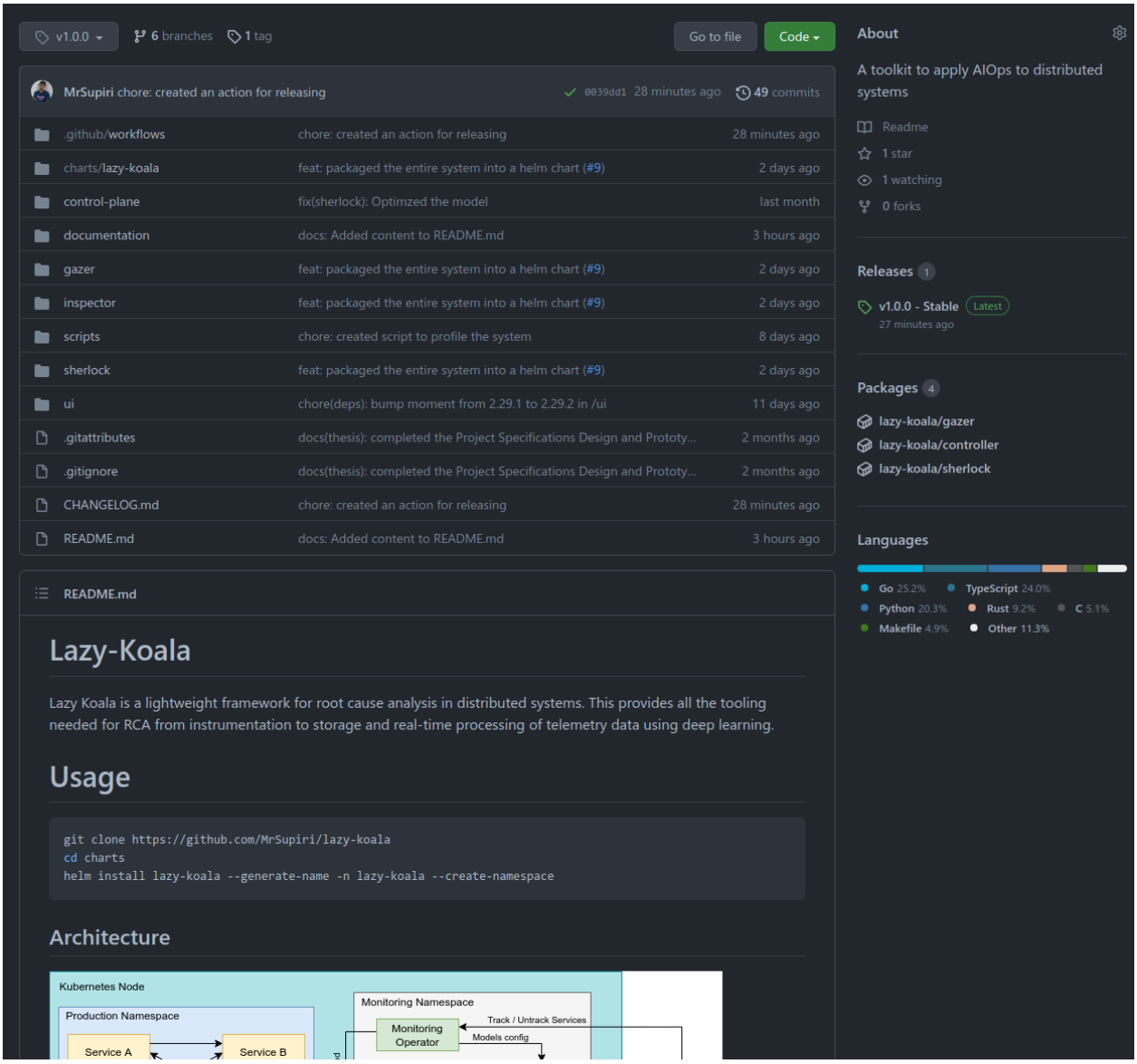


Figure I.1: Github Repository Structure (self-composed)

I.2 Conventional Commits

Conventional commits allow both humans and machines to easily understand the changes made to the code. It starts with and tag which indicates the type of change made. Following that the scope of the commit specifies which part of the code base code changed. Finally, the commit message is used to describe the change (*Conventional Commits*, n.d.).

```
* a238382 (HEAD -> main, origin/main, origin/HEAD) feat: packaged the entire system into a helm chart (#9)
|  27 files changed, 1417 insertions(+), 278 deletions(-)
* f2303f3 chore: created script to profile the system
|  2 files changed, 132 insertions(+)
* 9927e9c chore(deps): bump moment from 2.29.1 to 2.29.2 in /ui
|  2 files changed, 8 insertions(+), 8 deletions(-)
* dc6414a fix(sherlock): Optimzed the model
|  9 files changed, 543 insertions(+), 329 deletions(-)
* 8f1a791 fix(control-plane): Models not registering
|  6 files changed, 124 insertions(+), 17 deletions(-)
* 8911c9b feat(sherlock): Added the logic to normalize data frame
|  3 files changed, 48 insertions(+), 7 deletions(-)
* 9e52777 fix(gazer): Added backlog cleaner
|  1 file changed, 34 insertions(+), 22 deletions(-)
* cc964e9 fix(gazer): poll_kube_api exits after the first call
|  3 files changed, 40 insertions(+), 33 deletions(-)
* f0fe3a9 fix(sherlock): logged missing metrics
|  1 file changed, 1 insertion(+), 1 deletion(-)
* a46a587 chore: Added a data collection agent
|  8 files changed, 128 insertions(+), 15 deletions(-)
* 3e6e912 fix(inspector): Added missing Env file
|  2 files changed, 4 insertions(+), 2 deletions(-)
* 41f1195 fix(inspector): Typescript errors
|  6 files changed, 58 insertions(+), 235 deletions(-)
* db10b66 feat(inspector): Integrated proxy with kubernetes
|  7 files changed, 185 insertions(+), 54 deletions(-)
* 6b645bb feat(ui): Created the  visualization dashboard
|  5 files changed, 2138 insertions(+), 15 deletions(-)
* 6a14c90 Added proxy prom API
|  1 file changed, 13 insertions(+), 2 deletions(-)
* 0333c60 ci(inspector): Fixed a misconfigured path
|  2 files changed, 2 insertions(+), 13 deletions(-)
* 944aa13 feat(inspector): Created Settings View (#6)
|  31 files changed, 6038 insertions(+)
* 5b49dbf fix(sherlock): Moved poll_anomaly_scores sleep to async
|  3 files changed, 30 insertions(+), 15 deletions(-)
* 247ee28 fix(control-plane): Fixed a pointer mixmatch
|  2 files changed, 4 insertions(+), 4 deletions(-)
* d802f0a fix(control-plane): Fixed a bug in servings config generation
|  5 files changed, 35 insertions(+), 17 deletions(-)
:
```

Figure I.2: Use of Semantic Commits (self-composed)

I.3 Continuous Integration & Continuous Delivery Pipeline

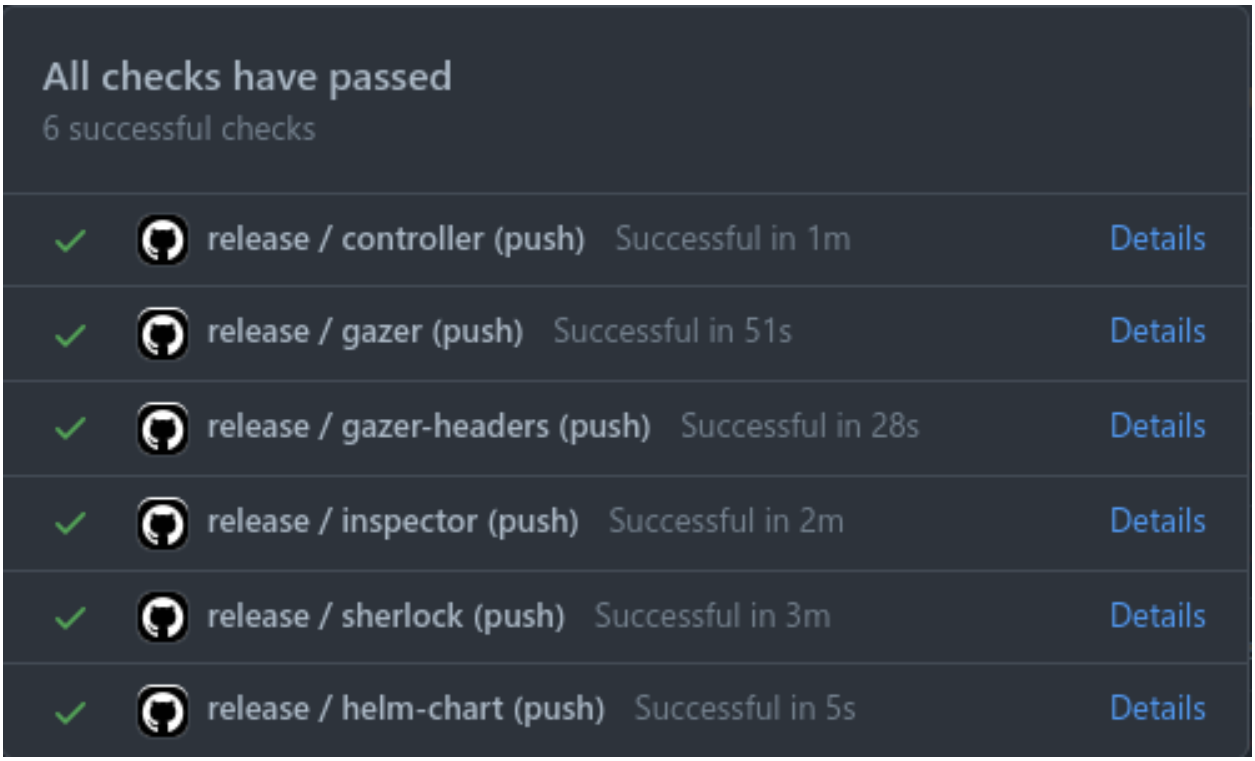


Figure I.3: Continuous Integration & Continuous Delivery Pipeline (self-composed)

I.4 Release Note

Figure I.4: Generated release note from the commit history (self-composed)