Bonsai

David Fischer

20th April 2022

A project report submitted in partial fulfilment for the degree of

BSc (Hons) Computing

Computer Science

University of Central Lancashire

Abstract

In the everchanging landscape of IT infrastructure, monitoring the health and performance of components is crucial to maintain steady uptime and service availability. Current monitoring systems can exhibit high latencies and complexity, potentially leading to an increased risk of system failure and a high amount of maintenance effort to keep each individual component running. This project's goal is the creation of a dynamic and scalable solution capable of collecting metrics from any source. Thus, "Bonsai" a minimalistic monitoring system was developed. The primary objective of Bonsai was the creation of a complete monitoring system that gets metrics from a host to a dashboard as fast as possible. To achieve this, a multitude of methodologies were considered and implemented. The system was designed with a minimalistic approach, reducing each component of traditional monitoring systems to be as atomic as possible. This greatly helped to reduce the complexity of the system, making further expansion or changes to the components easy to achieve. Service oriented architecture was used to isolate components from each other as much as possible. This further helped to increase the simplicity of the project, and also allows for the system to be used for other tasks than originally planned. An example of this would be deploying the system without a frontend, converting it from a full monitoring system to an ingress engine. Bonsai's achievements include the successful development of a complete monitoring system, featuring a set of example metrics exporters, an ingress server, a socket, and a frontend. Each component and the communication methodology between them were created to ensure rapid data processing, enabling near real-time visualisation of metrics from a server on a client's dashboard. Due to the systems approach to services, modification and expansion is a possibility, making it possible for custom tailored solutions to be implemented. Data exporters can be written in a broad range of languages, to collect virtually any metrics. The server, used for metric ingress, is able to handle a multitude of exporters concurrently pushing data into the system. The socket is able to serve multiple clients concurrently without impacting any other system components performance. The frontend provides a pleasant user experience, while still maintaining a high grade of functionality. Each component was tested in a deployment, which accurately resembles real world conditions, in which the system proved itself to be a viable option for monitoring.

Attestation

I understand the nature of plagiarism, and I am aware of the University's policy on this.

I certify that this document reports original work by me during my University project. I also confirm that I adhere to the University's legal and ethical guidelines for undergraduate projects in Computing.

Signature: [illegible]

Date: 24.04.2023

Acknowledgements
The author would like to thank the project supervisor Matthew Bates for his guidance and support throughout the work on this project.

Table of Contents

Abstract i

Attestation ii

Acknowledgements iii

Table of Contents iv

List of Figures ix

List of Tables x

List of Listings xi

1 Introduction 1

1.1 Rationale 1

1.2 Motivation 1

1.3 Goals 1

1.4 Challenges 1

1.4.1 Viability of a Minimalistic Monitoring System 1

1.4.2 Implementation of a Fast and Dynamic Data Transfer Standard 1

1.4.3 Learning Effort 2

1.4.4 Achievability of the Project's Scope 2

2 Background and Related Work 3

2.1 Introduction 3

2.2 Current Monitoring Practices 3

2.2.1 Introduction 3

2.2.2 Grafana 3

2.2.3 Time Series Database 4

2.2.4 Prometheus 4

2.2.5 Data Exporters 5

2.3 Technologies 6

2.3.1 Introduction 6

2.3.2 gRPC 6

2.3.3 RethinkDB 7

2.3.4 Socket.IO 7

2.3.5 Model View ViewModel 8

2.3.6 Docker 8

2.4 Methodologies 9

2.4.1 Introduction 9

2.4.2 Bonsai 9

2.4.3 Service Oriented Architecture 9

2.4.4 Monitoring Architecture 10

2.4.5 Data Exporters 10

2.4.6 Database Design 10
2.4.7 Communication between Exporter and Server 10
2.4.8 Querying Data 11
2.5 Summary 11
3 Project Planning 12
3.1 Introduction 12
3.2 Methodology 12
3.2.1 Prototyping 12
3.2.2 Incremental Development 12
3.2.3 Gantt Chart 12
3.2.4 Waterfall 12
3.2.5 Kanban 13
3.3 Requirements 13
3.4 Potential Solutions 14
3.5 Tools and Techniques 14
3.6 Legal, Social, and Ethical Issues 14
3.7 Summary 15
4 Design 16
4.1 Introduction 16
4.2 System Design 16
4.2.1 Service Oriented Architecture 16
4.2.2 Minimizing Dependencies 16
4.2.3 Minimizing Traffic 16
4.2.4 Exporter Data Format 16
4.2.5 Protocol Buffer Classes 17
4.2.6 Dynamic Protocol Buffers 18
4.3 Service Design 18
4.3.1 Planned Structure 18
4.3.2 Data Exporter 18
4.3.3 Server 18
4.3.4 Socket 19
4.3.5 Frontend 19
4.4 Docker 20
4.4.1 Docker Images 20
4.4.2 Docker Image Size 20
4.4.3 Docker Compose 20
4.4.4 Docker Compose Structure 20
4.5 User Interface Design 21

4.5.1 Views ... 21
4.5.2 Wireframes ... 21
4.5.3 Colour Scheme ... 22
4.6 Summary ... 22
5 Implementation ... 23
5.1 Introduction ... 23
5.2 Protocol Buffers ... 23
5.2.1 Introduction ... 23
5.2.2 Bonsai Proto File ... 23
5.3 Data Exporter ... 23
5.3.1 Introduction ... 23
5.3.2 Configuration ... 23
5.3.3 Exporter Classes ... 24
5.3.4 Pluggable Exporters ... 24
5.3.5 Entrypoint Script ... 25
5.3.6 Client Class ... 25
5.3.7 Server Communication ... 25
5.4 Server ... 26
5.4.1 Introduction ... 26
5.4.2 gRPC Service Implementation ... 26
5.4.3 Database Controller ... 26
5.4.4 Middleware ... 27
5.4.5 Reflection and Health Monitoring ... 28
5.4.6 TLS Implementation ... 28
5.5 Socket ... 28
5.5.1 Introduction ... 28
5.5.2 Database Connection Pool ... 28
5.5.3 Socket.IO ... 28
5.5.4 Socket Routes ... 29
5.5.5 Listener Implementation ... 29
5.5.6 REST API ... 30
5.5.7 REST Controller ... 30
5.6 Frontend ... 31
5.6.1 Introduction ... 31
5.6.2 Vue Router ... 31
5.6.3 Socket Communication ... 31
5.6.4 Home View ... 32
5.6.5 Dashboarding View ... 33

5.6.6 Node Graph View 34
5.6.7 Explore View 34
5.6.8 Reverse Proxy 35
5.7 Docker Implementation 36
5.7.1 Introduction 36
5.7.2 Container Structure 36
5.7.3 Docker Compose Deployment 36
5.7.4 Health Monitoring 37
5.8 Summary 37
6 Test Strategy 38
6.1 Introduction 38
6.2 Development Lifecycle 38
6.2.1 Development with Docker 38
6.2.2 Environment Based Configuration 38
6.3 Continuous Integration and Continuous Delivery 38
6.3.1 Introduction 38
6.3.2 GitHub Actions 39
6.3.3 GitHub Registry 39
6.3.4 Ansible 39
6.4 Testing Infrastructure 39
6.4.1 Introduction 39
6.4.2 Hybrid Cloud Infrastructure 40
6.4.3 Deployment 40
6.4.4 Conclusions 40
6.5 Stability Testing 40
6.5.1 Data Exporter Transfer 40
6.5.2 Resource Utilization 41
6.6 Functional Testing 41
6.6.1 Unit Testing 41
6.6.2 Integration Testing 42
6.6.3 System Testing 42
6.6.4 Acceptance Testing 42
6.7 Non-Functional Testing 42
6.7.1 Performance Testing 42
6.7.2 Scalability Testing 43
6.7.3 Usability Testing 43
6.7.4 Security Testing 43
6.8 Evaluation 44

6.9 Summary 44
7 Evaluation, Conclusions and Future Work 45
7.1 Project Objectives 45
7.2 Self-Evaluation 45
7.2.1 Author Motivation 45
7.2.2 Reflection on Prototyping 45
7.2.3 Reflection on Minimalism 45
7.2.4 Reflection on the Frontend 46
7.2.5 Reflection on the Quality of Code 46
7.2.6 Lessons Learned 46
7.3 Project Evaluation 46
7.3.1 Server 46
7.3.2 Data Exporters 46
7.3.3 Socket 46
7.3.4 Frontend 47
7.4 Applicability of Findings to the Commercial World 47
7.5 Conclusions 47
7.6 Future Work 47
7.6.1 Exporters 47
7.6.2 Dashboarding 47
7.6.3 Components 48
7.6.4 Comparison to other Systems 48
References 49
Appendix 1 – Project Proposal 53
Appendix 2 – Technical Plan 56
Appendix 3 – IOS Gyroscope Exporter 63

List of Figures

Figure 1: Grafana displaying a Dashboard of metrics from Node Exporter (Node Exporter Full, 2022) 3
Figure 2: Illustration of a simplified monitoring architecture built around Prometheus 5
Figure 3: Diagram describing a gRPC exchange (Introduction to GRPC, 2022) 6
Figure 4: Illustration of a Socket.IO emit event based on a RethinkDB listener (Broadcasting Events | Socket.IO, 2022) 8
Figure 5: Diagram giving an overview over the VueJS MVVM implementation (Getting Started - Vue.JS, 2022) 8
Figure 6: Simplified proposed structure of a monitoring architecture built around Bonsai 10
Figure 7: Example Kanban Board filled with Tasks related to this Project 13
Figure 8: Proposed Structure of a Data Exporter's JSON Formatted Output 17
Figure 9: Illustrated Diagram of a Proposed Protocol Buffer Structure 17
Figure 10: Illustration of a Potential Bonsai Data Exporter Class Diagram 18
Figure 11: Illustration of the Bonsai Servers Proposed Internal Processes 19
Figure 12: Illustration of the Bonsai Socket's Proposed Communication with the Frontend 19
Figure 13: Illustration of the Proposed Docker Compose File Structure 21
Figure 14: Wireframes of the four Planned Frontend Views 22
Figure 15: The BonsaiClient Class Implementation Illustrated as a Flowchart 25
Figure 16: Illustration of the TLS Connection Process 28
Figure 17: Illustration of a Web Application Implementing the Vue Router 31
Figure 18: Bonsai Frontend Home View 32
Figure 19: Bonsai Frontend Dashboard View 33
Figure 20: Example Dashboard Data Structure 34
Figure 21: Bonsai Frontend Explore View 35
Figure 22: Bonsai Frontend Node Graph View 40
Figure 23: Plot of a Data Exporter's Scraping and Transfer Time over a Period of 14 Days 41
Figure 24: Section of a Grafana Dashboard Observing the Bonsai Stack Deployment 41
Figure 25: Comparison of RethinkDB and MongoDB using Timed Insertions and Updates 42
Figure 26: Section of a Grafana Dashboard Observing a Grafana Stack Deployment 43
Figure 27: RethinkDB Admin Dashboard Displaying Statistics for the Metrics Table 43

List of Tables

Table 1: MoSCoW Method applied to each Requirement of the Projects Components.... 14

List of Listings

Listing 1: [bonsai.proto] Snippet of the Bonsai Proto Definition File 23
Listing 2: [report_config.yml] An Example of a Complete Bonsai Exporter Configuration 24
Listing 3: [BonsaiConfigLoader.py] Snippet of the BonsaiConfigLoader Class 24
Listing 4: [BonsaiClient.py] Snippet of the BonsaiClient Class to Demonstrate Request Construction 26
Listing 5: [BonsaiServer.py] Snippet of the BonsaiServer Class 26
Listing 6: [RethinkServerConnection.py] The RethinkServerConnection Class 27
Listing 7: [BonsaiServer.py] Simplified Snippet of the BonsaiServer Class PushMetrics Function 27
Listing 8: [index.js] Snippet of the Bonsai Socket Index File 29
Listing 9: [dashboardRoutes.js] Snippet of the Bonsai Socket Dashboard Route Definitions 30
Listing 10: [HomeView.vue] Simplified Snippet of the Script Section of the Home View ... 32
Listing 11: [HomeView.vue] Simplified Snippet of the Template Section of the Home View 33
Listing 12: [nginx.conf] Simplified NGINX Config Implemented by the Frontend Docker Container 35
Listing 13: [Dockerfile] Dockerfile Definitions of the Bonsai Server 36
Listing 14: [docker-compose.simple.yml] Simplified Version of this Project's Docker Compose File 37
Listing 15: [rethink.js/rethink.py] Example Snippet of Environment Based Configuration. 38
Listing 16: [publish_docker.yml] GitHub Actions Workflow which Builds and Publishes Containers 39

1 Introduction

1.1 Rationale

With the modern world of IT being an everchanging landscape from infrastructure to application design, effectively monitoring a system is becoming more and more challenging. As businesses rely on these systems, having insight into every aspect of an application, from the hardware it is running on, up to the application's health, is crucial. To name just some of the areas of importance, cloud monitoring, DevOps monitoring, end user monitoring, security monitoring, network monitoring, hardware monitoring, and software monitoring are all things to be considered when deploying a monitoring system (Fatema et al., 2014). There are many monitoring systems that fill some or all parts of this chain, some of which will be discussed in this report, however none fulfil the vision this project aims to achieve.

1.2 Motivation

The primary goal of this project was the creation of a monitoring system that is able to challenge current standards by standing out from other systems. The system that will be created in part of this report should be dynamic, in the sense that it can monitor any type of system, as long as a data exporter has been created for the required metrics, while being truly real-time, meaning that the transfer times between all system components will be made to be as efficient as possible. This vision should be fulfilled in a minimalistic sense, where every system component is created to be as atomic as possible, meaning that any functionality going above the components primary functionality should be made to be another component. All of these requirements are not fully supported by any existing monitoring system, giving this project a space to fill in the modern monitoring landscape.

1.3 Goals

This project aims to achieve a complete monitoring system fulfilling all requirements laid out in the previous section. To implement the system, multiple components will need to be researched and created. A database will need to be selected, which is able to store and deliver dynamic data in real time. A server capable of receiving and parsing data from multiple data exporters concurrently will need to be built. Data exporters, as well as a standard for communication between the exporters and the server, will need to be created. To make the data presentable a frontend with dashboarding capabilities will need to be implemented. Another component, which is capable of listening for changes in the database and relaying them to the frontend in some way, will also need to be created. Finally, the entire project should be deployable in any existing infrastructure by making the individual components distributable in some form of package.

1.4 Challenges

This section will cover the initial set of challenges which present themselves when planning to create a system like this.

1.4.1 Viability of a Minimalistic Monitoring System

There are a variety of existing monitoring systems being used across all industries, most of them being focused on achieving a high amount of data retention. To keep this project's minimalistic vision, the only metrics saved in the database will be the last set received from all data exporters. While this downside could be circumvented with the creation of a component that is capable saving metrics to be used for historical analysis, it is not planned for the initial version of this project.

1.4.2 Implementation of a Fast and Dynamic Data Transfer Standard

There are a multitude of data transfer standards on the application layer, each of which come with different up and downsides. Some examples of this would be REST, gRPC, and GraphQL, all of which are standards implemented to provide some kind of application programming interface for other software. A REST API would fit the dynamic vision of this project perfectly, however its performance in regards to transfer speeds, especially when compared to gRPC, would be a downside (Bolanowski et al., 2022).

1.4.3 Learning Effort

As this project aims to cover a multitude of different areas within computing such as systems design, systems administration, software engineering, network management, performance optimization, as well as frontend design, the learning curve could prove to be a challenge. To circumvent this, a vast amount of research will be required to ensure all components of the system are implemented to a high standard.

1.4.4 Achievability of the Project's Scope

Due to this project's large scope in regard to the number of components which need to be created in order for the monitoring system to be fully functional, the implementation will heavily rely on proper planning with regards to each features specific needs. As the implementation will cover many different languages and technologies, thorough research as well as early prototyping will need to be done to determine which components will require more attention and therefore implementation effort. In addition to the development effort, all components will need to be packaged some form which allows distribution, to ensure deployments can be created easily.

2 Background and Related Work

2.1 Introduction

Monitoring has been a quintessential part of modern IT Systems for many years (Schlossnagle, 2017). The ability to detect outages, irregularities, or intrusions into a productive system can be critical in keeping services up and running. Monitoring serves as the entry point to many IT related processes, most commonly being alerting systems, which enable teams to quickly get notified of potential issues with their infrastructure.

This project aims to develop a monitoring system capable of receiving data from various different sources and storing it in real time. This chapter reviews related work to the project, the influences taken from existing monitoring systems, and the methodologies that will be used to implement and develop the system.

2.2 Current Monitoring Practices

2.2.1 Introduction

This section aims to discuss the current trends which are prevalent in observing and monitoring IT systems. Most monitoring systems implement multiple software applications, protocols, and data storage solutions to achieve their goal (Cerny et al., 2022). Currently, one of the most prevalent analysis and visualisation platforms is Grafana, which serves as the centrepiece to many companies monitoring solutions (Grafana, 2022). Grafana, an open source web application first released by Torkel Ödengaard in 2014, supports multiple types of databases, the most frequent being time series databases (Goldschmidt et al., 2014). Continuing, this section offers an insight into how a modern monitoring stack is built from data collection to visualization using Grafana and Prometheus as the core components.

2.2.2 Grafana

As previously mentioned, Grafana is a fully featured observability platform that is able to process and visualize data from different types of sources by implementing support for different databases and their respective querying languages. Due to the project being open source and backed by Grafana Labs, the project enjoys rolling releases with few weeks between sub releases (Grafana Labs, 2022). Grafana stands out next to other tools due to its expandability and excellent ability to process and visualize time-series data, which is more closely explained in Section 2.2.3. Thanks to the open source approach, Grafana has support for various community plugins and extensions, ranging from custom data sources to visualization methods (Mohammed et al., 2022). All of these factors play into Grafana's current market leading position in its specific use case.

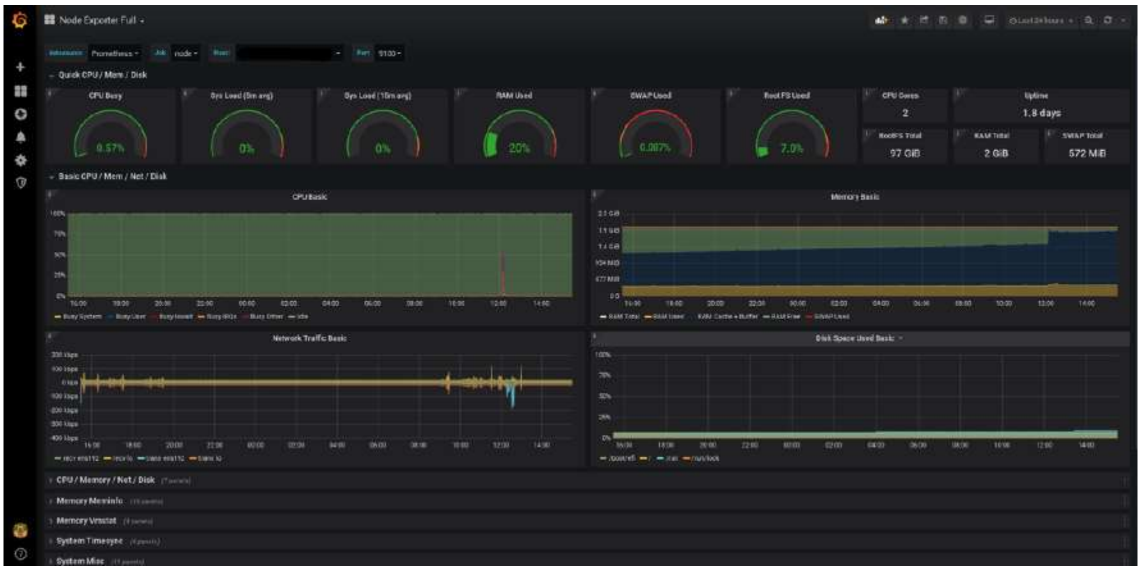


Figure 1: Grafana displaying a Dashboard of metrics from Node Exporter (Node Exporter Full, 2022)

2.2.3 Time Series Database

A time series database, often abbreviated as TSDB, is a software system that is built to store and serve data, which is structured in associated pairs of times and values. To help illustrate the concept of time series databases for monitoring, one can imagine a dictionary, which is a data structure made of keys and their corresponding values. In this context, the keys used to access data are timestamps, that indicate when the entry was made, and the values are the metrics collected at that moment in time. TSDBs differ from relational databases in nature, due to their key and value pair approach, usually featuring no referential concepts when storing data (Harpreet et al., 2013). This approach is inherited from the NoSQL database structure, which TSDBs or more specifically key-value databases adhere to. NoSQL, or in other words either "non-SQL" or "non-relational" databases mostly apply the same methodologies relational databases do when it comes to inserting or selecting data, with the key difference being how the data is stored. Relational databases use a tabular approach with connections being drawn between tables, while NoSQL systems discard this system and take an approach most suitable for their use case.

A major advantage of time series databases is their sheer efficiency and speed when compared to relational databases, as TSDBs implement various features to improve performance (Shah et al., 2022). In part due to these advantages when working with time series data, TSDBs are an essential part of long term monitoring in modern monitoring architectures. TSDBs are not only ideal for monitoring applications, but also for general data collection. Their ability to measure change over time is unmatched by other database structures, thus they are implemented across many industries. An example of this would be their use in financial markets as stock prices constantly fluctuate, creating vast amounts of data to be stored across different points in time.

2.2.4 Prometheus

Prometheus is an open source monitoring system that implements a TSDB to collect and process data (Prometheus, 2022). Like Grafana, Prometheus is open source software that receives rolling updates that provide security updates and other improvements. While not offering much expandability or scalability by itself, Prometheus is built to ingress multiple types of metrics (Birje & Bulla, 2019). This further solidifies its standing in current monitoring practices as data exporters can be built and implemented around almost any existing infrastructure, which requires monitoring, observation, and alerting.

Prometheus scrapes data by sending HTTP requests to data exporter endpoints. The request interval and location of these endpoints are configured on the Prometheus host. The returned data is formatted using a predefined structure, which is used to parse and write metrics into the built in Prometheus TSDB. Data exporters are more thoroughly explored in Section 0.

As mentioned before a full monitoring architecture relies on multiple different tools. An architecture built around Prometheus would include:

- Prometheus to scrape and store metrics
- Multiple separate data exporters running on hosts to be monitored
- Grafana to visualize metrics
- Additional software, like Alertmanager, to send notifications if irregularities are detected

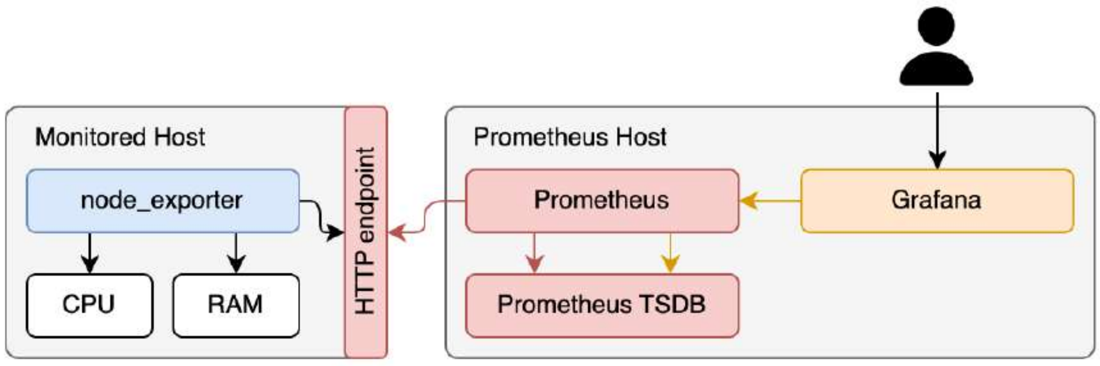


Figure 2: Illustration of a simplified monitoring architecture built around Prometheus

As can be seen in Figure 2, Prometheus collects data by querying data exporters that open HTTP endpoints on their respective hosts. The data from the HTTP request is then written into the built-in Prometheus TSDB. To facilitate communication between Prometheus and other software like Grafana or Alertmanager, Prometheus implements PromQL, a functional query language built for Prometheus (Introduction to PromQL, the Prometheus Query Language, 2020). PromQL lets other software interact with Prometheus by providing a way to select and aggregate the stored time series data in real time. PromQL benefits from a simple syntax, while still maintaining powerful data aggregation functionalities. Querying languages in general are an essential part of monitoring systems as they allow users to extract and analyse the vast amounts of collected data.

2.2.5 Data Exporters

Data exporters, as their name suggest, serve as the collectors in a monitoring architecture. Prometheus provides libraries that enable developers to build exporters in multiple languages. These libraries ensure that exporters implement the standards set by Prometheus, as they need to follow a specific type of formatting when serving collected metrics. Prometheus has a vast community of developers writing exporters for different applications ranging from simple host metrics to databases to networking appliances. An example of this would be node exporter, a data exporter developed by the Prometheus community, which collects hardware and operating system metrics exposed by machines running UNIX-like systems (Node Exporter, 2013/2022).

As mentioned previously, Prometheus queries endpoints exposed by exporters using HTTP. The return to these queries has a predefined format that every exporter must adhere to. Each metric point returned from the exporters is headed by a "HELP" and "TYPE" expression. These are saved by Prometheus to assist in categorizing metric types and providing descriptions to users. As mentioned, the labels and descriptions provided by the exporters also factor into the PromQL querying language, enabling it to perform complex data aggregation while retaining a simple syntax.

By default, most data exporters built for Prometheus, including node exporter, expose their endpoints to plain HTTP request without any form of encryption or authentication. While this can be reconfigured or circumvented using proxies, it adds time an engineer has to spend to secure the monitoring system. A study done on the security concerns of open endpoints on IoT devices and the security concerns stemming from them relates very closely to these exporters (Tedeschi et al., 2019). Even on a closed network implementing measures to secure data exporters is essential. Depending on the nature of data being collected, serious implications could stem from exporters being accessible to any bad actor using means as simple as a web browser.

2.3 Technologies

2.3.1 Introduction

This chapter will introduce technologies that, once researched, have the potential to be used in order to implement the functionality this project aims to achieve. These technologies will be discussed in relation to current and future monitoring standards, highlighting their strengths and potential limitations. The subsections that follow will provide an in-depth look at each of these technologies, discussing how they work and how they can be integrated into the project's overall design.

2.3.2 gRPC

gRPC is a high performance, open source universal RPC framework. RPC, standing for remote procedure call, enables programs to call a procedure from a different address space, as if it were a local procedure call (GRPC, 2022). gRPC can run in any environment, to connect services across networks with support for authentication, tracing, and load balancing (Matos et al., 2021). To facilitate its communication method, gRPC implements Protocol Buffers to transfer data between services. Developed by Google, Protocol Buffers are a mechanism to serialize structured data (Protocol Buffers, 2022). Protocol Buffers allow complex data structures to be predefined across multiple languages, thus making the serialized data language neutral. It is therefore a great utility when building distributed systems that span across multiple languages or even networks.

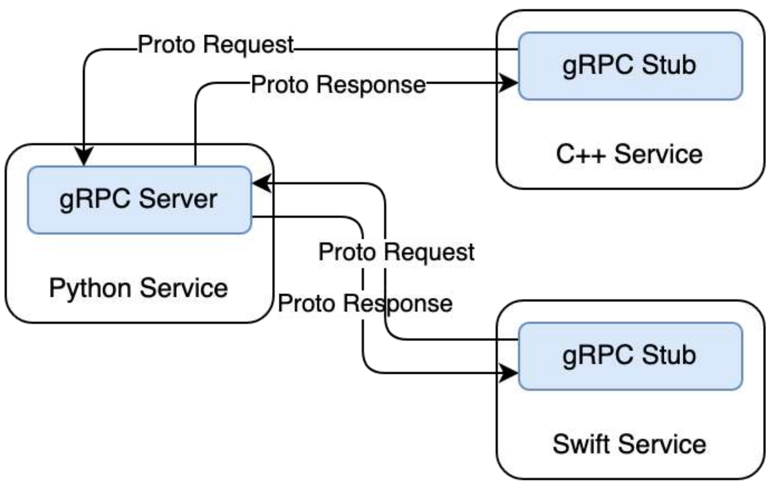


Figure 3: Diagram describing a gRPC exchange (Introduction to GRPC, 2022)

A major concern in any software system are its performance and speed. Compared to other programming interfaces, in relation to speed gRPC generally outperforms other methods like REST APIs or GraphQL (Śliwa & Pańczyk, 2021); (Chamas et al., 2017). Due to the implementation of Protocol Buffers, gRPC is able to standardise communication between services, vastly improving the development experience, as there is no uncertainty in what data is returned from which method. gRPC achieves its high performance due to the data being transferred in a serialized format, thus already implementing a layer of compression, and improving efficiency. It is therefore a great technology to facilitate data transfer and communication between multiple services and systems. Thus, in relation to monitoring, gRPC offers multiple advantages over other data transfer methods making it a standout choice for any data transfer application.

In monitoring, a constraint that an implementation using gRPC might face is related to its use of Protocol Buffers and predefined data structures. Monitoring, being dynamic in nature, usually requires multiple different definitions for metrics that depend on the type of exporter being used. There are multiple ways to get around this limitation posed by the Protocol Buffer data structures.

Protocol Buffers offer many predefined data types, most of which are defined in a similar manner to programming languages such as "double" or "float" which once transferred would take on the same shape in Python. Within Protocol Buffer messages, data types can be defined either as singular fields, lists, or dictionaries. Messages can also be used as a data type within other messages. Using this approach, a message for a metric message definition could be created which are stored as a list within another message. This would however still run into issues as a static message definition would limit the types of data that can be transferred. Another method would be using the built in "bytes" data type, as it allows any arbitrary sequence of bytes to be transferred over gRPC. This would enable services to transfer dynamic data without any structure constraints while still using gRPC and implementing its other advantages.

2.3.3 RethinkDB

RethinkDB is a free and open source document oriented database, which is marketed as "The open source database for the realtime web." Inherently, document oriented databases like RethinkDB are a subclass of the already mentioned key and value approach to data storage. Comparing document oriented databases with time series databases is therefore a possibility, as they both build upon the NoSQL database methodology. A key concept of document oriented database systems are documents, which can be compared to objects when thinking about data in programming. While the concept differs between different implementations, documents are usually data stored in a standardised format or encoding, including YAML, JSON, XML, and others, each coming with certain advantages and disadvantages (Rianto et al., 2021). This document focused approach offers many advantages when compared to relational databases, one of the key differences being the speed of operations (Chickerur et al., 2015). Another advantage over relational databases is the ability to save dynamic data, which relates back to monitoring. Monitoring is quite dynamic in nature due to the everchanging IT landscape, thus a database system capable of storing dynamic data is essential to most monitoring architectures.

Looking further into the features offered by RethinkDB, its standout feature, next to its superior speed, among other document oriented databases, is its approach to serving data in real time with support for change feeds (Suma & Alqurashi, 2019). In regular monitoring approaches, data is polled either by the user or software which can result in a slower application or the inability to scale (Van de Vyvere et al., 2020). RethinkDB aims to solve this by pushing data to applications. These listeners reduce engineering efforts on the developer's end while still offering vast amounts of functionality.

In addition to ReQL, its querying language, RethinkDB has extensive official APIs for JavaScript, Python, Ruby, and Java, with more languages being supported through members of its community. A real time change feed can be implemented using only a few lines of code, which makes working with RethinkDB an intuitive experience.

2.3.4 Socket.IO

Socket.IO is a real time library for web applications that implements communication between clients using a web browser, and a server. Similar to RethinkDB, Socket.IO is specialized to be event driven in nature. To facilitate communication, Socket.IO uses the WebSocket protocol, which is a communication protocol that implements bidirectional messaging over a single TCP connection. In contrast to REST APIs, which are accessed using a request which results in one response, WebSockets enable applications to receive streams of data, which opens up opportunities for many different application types (Singh et al., 2021). There are two libraries offered by Socket.IO, one for web applications and one for servers. These libraries are well documented and integrate well with other libraries, for example, the libraries offered by RethinkDB. An example usage of Socket.IO with RethinkDB would be a chat application, in which messages are stored within a database. Once a new message is written into the database, a listener on the table can hand the data to a Socket.IO instance, which then sends it to all connected clients in real time, as can be seen in Figure 4. The approach to interfacing with web applications introduced by Socket.IO relates closely to monitoring, especially monitoring focused on real time, as metrics can be delivered as streams to an observing client as soon as they are collected.

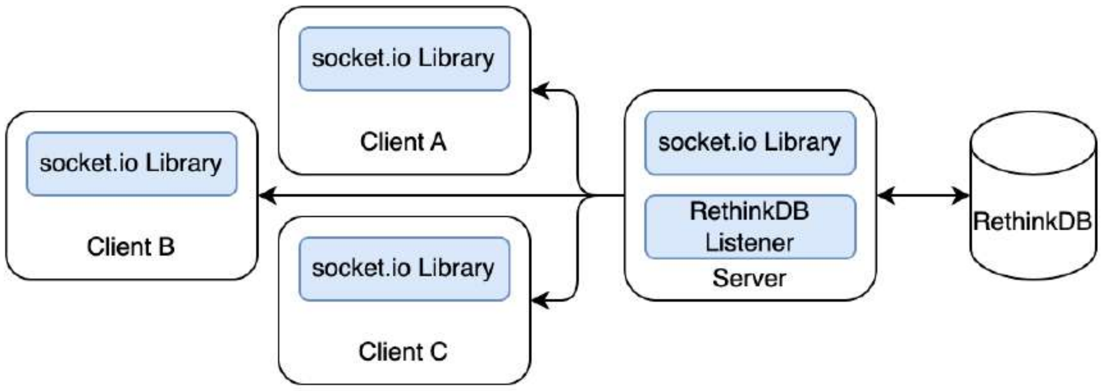


Figure 4: Illustration of a Socket.IO emit event based on a RethinkDB listener (Broadcasting Events | Socket.IO, 2022)

2.3.5 Model View ViewModel

Modern web applications heavily rely on background logic to keep them running, be it backend infrastructure for data delivery or other functionalities. Web application frameworks like VueJS, Angular or React implement different design patterns to achieve this. VueJS, a library for building interactive web interfaces, implements their variant of the Model View ViewModel (MVVM) design pattern. MVVM is a software design pattern built around the concept of separating the program logic and user interface (Syromiatnikov & Weyns, 2014). VueJS has a specific focus on the ViewModel layer of the MVVM pattern as can be seen in Figure 5. The Document Object Model (DOM) is a programming interface for web documents, in simpler terms, the storage for what a user sees when accessing a web application. Using VueJS, the DOM can be accessed and modified at any time through the ViewModel layer, based on either user interaction or other factors like responses from API endpoints or sockets. One of the key benefits of the MVVM patter with VueJS is that it allows for better separation of concerns, with each layer of the model only having a small number of total tasks. The View layer only needs to render the user interface, the Model layer only stores data, while the ViewModel layer handles data updates and interactions. This approach provides improved code maintainability and better performance while remaining simple to grasp and to implement code for. In relation to monitoring, frontend interfaces are what allows users to draw conclusions from their collected data. Implementing a feature rich web application is therefore a key element of modern monitoring.

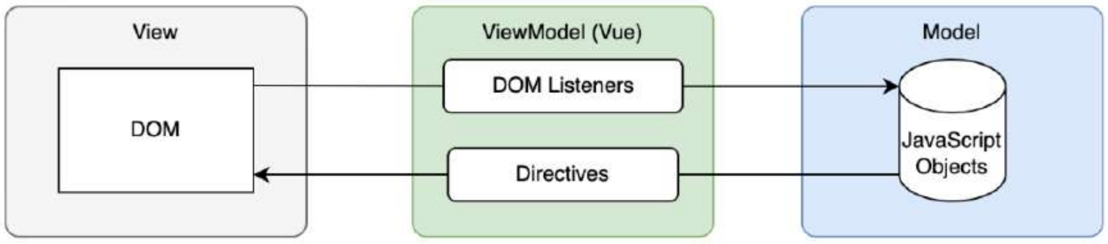


Figure 5: Diagram giving an overview over the VueJS MVVM implementation (Getting Started - Vue.JS, 2022)

2.3.6 Docker

Docker is a tool that allows developers to easily deploy, run, and manage applications by using containers (*Docker*, 2022). Containers allow developers to package applications, including all system critical components, like libraries, dependencies, and packages. These pre-built containers can then be distributed as one package. Docker enables these packaged applications to run on any underlying infrastructure, similar to Hypervisors but with a reduced number of layers between the physical host and the application. Using Docker during development enables developers to quickly test and deploy their applications without needing to be concerned about consistency across environments (Boettiger, 2015). Docker is also able to manage a plethora of other aspects applications might require. Specific networking can be setup to connect Docker containers with specific rules to control traffic. Volumes can be created for Docker applications to store data upon. Rules can be defined for what ports of a Docker container are accessible from outside. Docker, like many other technologies or software mentioned in this Chapter, benefits from a vast community of developers and companies that package their applications and distribute them freely.

To further support this notion of consistency, docker-compose is a tool to run Docker applications that require multiple containers. To achieve this, docker-compose uses YAML files that define what containers are to be deployed and how they are interconnected and configured. The main advantage of docker-compose is its ability to spin up complete testing or even production environments with a single command, including the main application and any additional services like databases or workers (Ibrahim et al., 2021). These deployments are as mentioned, consistent across multiple environments, so teams can be guaranteed the same results even when working with different operating systems. This is especially critical for monitoring systems, as systems built around multiple services and applications require specific conditions to enable communication between them, which can be vastly simplified by using docker-compose.

2.4 Methodologies

2.4.1 Introduction

This section will go into further detail on how this project will implement the aforementioned technologies and what influences have been taken from the existing systems. In addition, design methodologies will be discussed and how they factor into the projects vision.

2.4.2 Bonsai

The Japanese art of bonsai involves growing small trees in pots, which has been derived from the Chinese art of penjing. Penjing focuses on creating miniature landscapes in pots that resemble real-life scenery, while bonsai only seeks to replicate the shape of real trees on a smaller scale. In the same way, this project has been named "bonsai" to refer to its nature being small and compact, unlike other solutions that often bring many features that may be unnecessary for the planned application. "Bonsai" is focused on creating an elegant solution, much like the art of bonsai, to contrast other modern monitoring systems. Other software applications have also chosen the name bonsai, either to imply design philosophies or to show relation to trees (Lopes et al., 2015). Both of the mentioned aspects relate to this project.

2.4.3 Service Oriented Architecture

Service oriented architecture (SOA) is a software design methodology which defines interactions between independent and modular services (Laskey & Laskey, 2009). The main advantage that stems from a SOA approach is the ability to build complex systems that require multiple services and applications while still retaining an overview and maintainability. Services, as defined by SOA are usually self-contained and can be deployed independently from other elements. This approach results in the creation of flexible and reusable applications and services that easily integrate with one another, even giving certain services the chance to be used in a different context.

Docker, which has been discussed in Section 2.3.6, is a near perfect match for the SOA methodology, as it can be used to deploy and manage the services that make up the application (Peinl et al., 2016). Thanks to Docker, single services can be restarted, without impacting overall performance, or automatically scaled up or down to meet performance requirements. These factors can make it easier to manage applications that span across multiple services while improving overall reliability and performance.

2.4.4 Monitoring Architecture

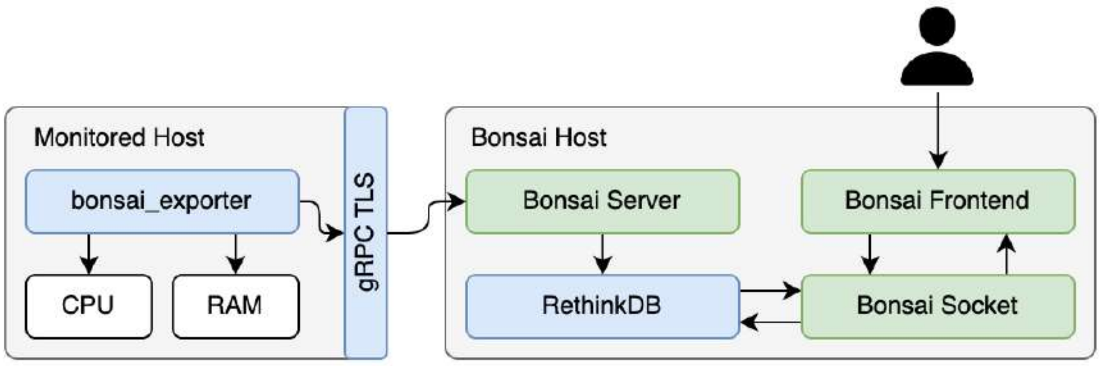


Figure 6: Simplified proposed structure of a monitoring architecture built around Bonsai

Comparing Figure 6 with Figure 2, there are a few key differences. Going from left to right, the project will implement gRPC over TLS to push metrics into the system, making it event driven (Maréchaux, 2006). This is in contrast to the Prometheus architecture, in which Prometheus itself queries the exporters. In this proposed structure, data received by the Bonsai Server is written into the RethinkDB, similar to the approach Prometheus takes. Unlike Prometheus, the Bonsai Server itself doesn't offer any way to query the database, as interfacing is handled using a separate service, called "Bonsai Socket" in the figure. Finally, user interaction with the monitoring system will be handled over a specially built web application that communicates with the socket. All of these proposed services are explained in further detail in the following subsections.

2.4.5 Data Exporters

As already mentioned, the entire system will be built around an event driven architecture as opposed to the more static approach implemented by Prometheus. The key to this architecture is the data exporters as they are the initial point data is collected in any monitoring system. One of the greatest struggles that can be faced while working with Prometheus is debugging exporters. There are two primary points of failure, in the exporters themselves, and the Prometheus configuration. This project aims to circumvent this by making exporters be specifically built around being as independent as possible from the main system. This approach also allows exporters to be added to the system without reloading any configurations on the monitoring host.

2.4.6 Database Design

This project aims to implement RethinkDB, a database system which has already been introduced in Section 2.3.3. Multiple different approaches can be taken within the bounds of a NoSQL database regarding table design. As the project aims to continually overwrite the metrics that are stored within the database with update operations, an approach that separates as much static information from the metrics as possible will need to be implemented. Update operations generally take more time than insertions or deletions across any NoSQL database implementation (Reichardt et al., 2021). To circumvent this and save time per update operation, a structure will need to be implemented that disconnects static metrics like hostnames and labels from the everchanging metrics. This can be achieved by borrowing the relational approach and utilizing two separate tables. Even though NoSQL databases generally do not implement relational structures, they are still able to replicate relations by using reference relationships (Sevilla Ruiz et al., 2015). Reference relationships are implemented by having one document store a reference to another document that contains related data. This is similar to the way a foreign key would be implemented in a relational database to establish connections between two tables.

2.4.7 Communication between Exporter and Server

Similar in nature to the database approach, the communication between data exporters and the server will need to be implemented in a way that keeps transfer size to a minimum to ensure speed and efficiency. The simplest way to implement this would be having data exporters register themselves to the server by sending over essential information like their hostname, exporter name and related labels. This also opens up the possibility to add a method of authentication before an exporter is allowed to store data in the system. The first layer of authentication would stem from gRPC, which has been discussed in Section 2.3.2, as a way to transfer data efficiently and securely between multiple services. Having the metadata separated from the metrics in this way would also help with the aforementioned database implementation as the metrics received from the exporters could be stored within the database without any major processing or modifications on the server side. In addition, this would also keep general network congestion to a minimum.

2.4.8 Querying Data

To follow the Service Oriented Architecture approach, the service responsible for querying data from the system will need to be built in a way that enables other services to easily interface with it. In other terms, the service will need to be conceptualized without keeping only the narrow view of this project's frontend in mind. Socket.IO, explained in Section 2.3.4, can aid this approach as a Socket.IO server can be accessed using multiple different clients and languages. To achieve this project's goal of becoming a fully featured monitoring system the querying service will need to fulfil a variety of tasks. Some examples of this would be querying metrics from hosts with specific labels or exporters, querying metrics from singular hosts, querying metrics from hosts that share certain similarities, and many more. There are many options available that can aid in implementing this, from querying languages to routes (Roy-Hubara & Sturm, 2020). Overall, the key to building a successful querying service within an SOA approach is to design it with flexibility and accessibility in mind, so that it can be easily integrated with other services and used to fulfil a wide range of querying tasks.

2.5 Summary

Development of this project will heavily rely on using the insights gained during this review of related works. In part of this chapter, modern monitoring trends have been discussed from their methodologies to potential downfalls. Technologies have been introduced that, when implemented correctly, will aid the project circumvent issues that other monitoring architectures face. The implementation will be supported by the design techniques mentioned in part of Section 2.4. Due to the broad discussion of technologies, the project also gains the advantage of not being tied to specific software mentioned within this chapter. An example of this would be RethinkDB not meeting performance targets, therefore it can be replaced with a different NoSQL database without any facing major time loss. Applying everything that has been mentioned will greatly support the projects development, even giving it the potential of going beyond the originally planned scope and features.

3 Project Planning

3.1 Introduction

As this projects scope is vast in terms of what services will have to be implemented to result in a fully functioning artefact, proper planning and implementation are essential. This section will cover multiple project management tools, which will be used to achieve a timely completion of every individual element, and therefore the project as a whole.

3.2 Methodology

3.2.1 Prototyping

To ensure enough time is allocated for each component and function of the project, early prototyping was done for the more complex tasks. This includes testing the feasibility of all technologies discussed in Chapter 2 for their planned implementations. Some examples of which functionalities would benefit from early prototyping would be the database, the communication method between the server and exporter, and the communication method between the socket and frontend. For the database a selection of NoSQL databases will be tested, primarily to examine their individual speeds and ease of implementation in relation to actively monitoring them for changes to documents. Lessons learned by prototyping the metric communication method will help estimate the time it will take to establish a system as well as a fitting database design. Finally, the applicability of a web socket for data streams will be tested both for its efficiency and applicability for a project of this nature. Completing this testing early using prototypes and small test applications, created solely for a single task will not only help with the estimation of timeframes for tasks, but it also provides a starting point for the final implementation, as well as establishing that the solutions are fit for their planned goals.

3.2.2 Incremental Development

As each component has a dependency on at least one other component functioning, each component will be implemented in this order. Using this bottom-up approach will guarantee that each component is implemented to at least the most minimal functioning state before work on the projects other system is started. This will also allow for more rapid adaptability during development, should any issues arise, which were not discovered during prototyping. It also helps isolate issues to the individual components, as opposed to them affecting the entire project. Another benefit to this approach is the early delivery of testable functioning systems, which can be refined as the project grows and more components are periodically developed and appended. Finally, breaking the project down this way ensures that development cycles only focus on one part of the project, as opposed to developing multiple components in parallel, splitting focus between them, which could result in implementations of lesser quality.

3.2.3 Gantt Chart

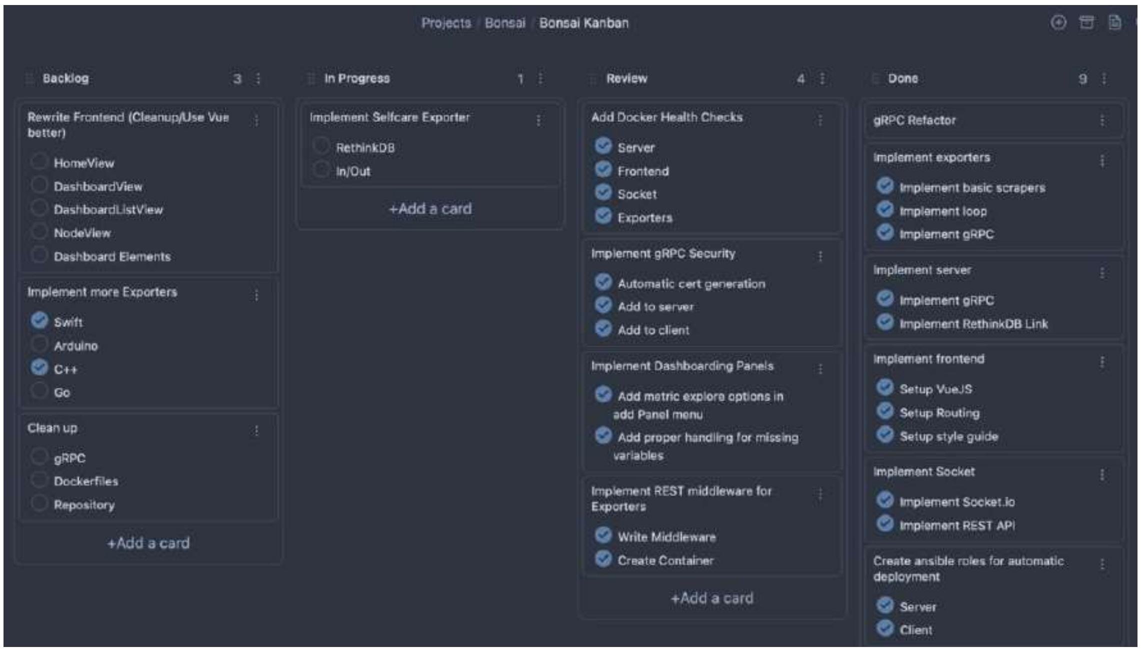
Each individual element of the project was broken down into multiple major features and given an estimation of how much time it would take to complete. These estimations were based on the aforementioned lessons learned from the prototyping, as well as the approach to incremental development. The tasks are then laid out using a Gantt chart, which results in a realistic plan on when to work on elements of the project. This project will choose the "waterfall approach" for its management, as with only one project member working on the implementation more advanced project management techniques will only add to the workload. A Gantt chart also supports this choice, as the tasks are already laid out in a form that resembles work under the waterfall approach. The Gantt chart used for this project can be seen in Appendix 2 – Technical Plan.

3.2.4 Waterfall

The waterfall project management approach describes a sequential order in which elements of the project are completed before moving on to the next phase. As discussed in Section 3.2.2, the incremental development method will be used for this project, thus making waterfall a good fit. The first stage of the waterfall approach is gathering requirements, which, after a system design phase, are then ordered into a list of timeframes in which tasks will have to be completed. Gantt lends itself well to waterfall, as it provides a visual overview of the individual tasks, as well as their deadlines. A visual similarity between a Gantt chart and an actual waterfall could also be mentioned, as tasks within a Gantt chart are ordered in a similar nature to what a waterfall might look like. The project will aim to deliver a prototype upon completion of every major task, to ensure testing for each component can be completed at an early stage. One of the drawbacks of the waterfall approach is revisiting a completed task, as the timeframe for the task is already over, thus sacrificing time which could be spent on another task (Thesing et al., 2021).

3.2.5 Kanban

To support the Gannt chart and give each task further details a Kanban board will be used. Kanban, coming from the Japanese word for "Card" is a visual system to manage workflows and optimize workflows in many industries (Ahmad et al., 2013). In Kanban, a board with cards is used to explain and distribute tasks. The board is usually split into multiple sections, which each represent a cards status. This project will use four sections; "Backlog" for outstanding tasks, "In Progress" for tasks which are currently being worked on, "Review", for complete, but untested tasks, and "Done" for complete tasks. Cards are then moved along these sections to categorize and manage workloads. Figure 7 is an example of the Kanban board used during development of this project, cards are titled after steps from the Gannt chart. Cards were also used to define tasks more granularly, by giving each feature smaller subtasks.



Projects / Bonsai / Bonsai Kanban

Backlog 3

Rewrite Frontend (Cleanup/Use Vue better)
- ☐ HomeView
- ☐ DashboardView
- ☐ DashboardListView
- ☐ NodeView
- ☐ Dashboard Elements

Implement more Exporters
- ✓ Swift
- ☐ Arduino
- ✓ C++
- ☐ Go

Clean up
- ☐ gRPC
- ☐ Dockerfiles
- ☐ Repository

+Add a card

In Progress 1

Implement Selfcare Exporter
- ☐ RethinkDB
- ☐ In/Out

+Add a card

Review 4

Add Docker Health Checks
- ✓ Server
- ✓ Frontend
- ✓ Socket
- ✓ Exporters

Implement gRPC Security
- ✓ Automatic cert generation
- ✓ Add to server
- ✓ Add to client

Implement Dashboarding Panels
- ✓ Add metric explore options in add Panel menu
- ✓ Add proper handling for missing variables

Implement REST middleware for Exporters
- ✓ Write Middleware
- ✓ Create Container

+Add a card

Done 9

gRPC Refactor

Implement exporters
- ✓ Implement basic scrapers
- ✓ Implement loop
- ✓ Implement gRPC

Implement server
- ✓ Implement gRPC
- ✓ Implement RethinkDB Link

Implement frontend
- ✓ Setup VueJS
- ✓ Setup Routing
- ✓ Setup style guide

Implement Socket
- ✓ Implement Socket.io
- ✓ Implement REST API

Create ansible roles for automatic deployment
- ✓ Server
- ✓ Client

Figure 7: Example Kanban Board filled with Tasks related to this Project

3.3 Requirements

This project has many individual and inherently different requirements, each of which will need to be completed in order to deliver a fully functioning artifact. To name the components, each with their most important requirements, a data exporter capable of collecting and sending metrics, a server capable of receiving and storing metrics in a database, a socket capable of listening for changes in the database to pass them to a client, and a frontend to interface with the socket and provide dashboarding for the metrics. While these requirements would result in a functioning system, the project has its aim set above them. To prioritize requirements, the MoSCoW method was applied to the tasks identified in the Gantt chart. MoSCoW, standing for Must, Should, Could, and Will not, is a system to categorize requirements by their importance to guide the projects implementation. The categorizations for each component of this project can be seen in Table 1.

	Database	**Server**	**Data Exporters**	**Socket**	**Frontend**
Must	Dynamic storage Efficient Communication	Communication standard implementation Pass information to database	Communication standard implementation Set of example exporter classes	Web Socket server Listeners on database for metric streams	Web Socket client Dashboarding capabilities
Should	Built-in Listeners	Efficiently parse incoming metrics	Dynamic addition of new exporter classes	Connection pool to database Support concurrent connections	Explore panel Node graph panel
Could		Connection pool to database for more efficient communication	Configuration file support Implementation in multiple languages	REST API for static communication	Custom dashboarding capabilities
Will not	Persistent metric storage				Alerting

Table 1: MoSCoW Method applied to each Requirement of the Projects Components

3.4 Potential Solutions

There are multiple potential solutions for each component and feature this project aims to achieve. The selected database will need to be able to store dynamic data, therefore relational databases will not be applicable for this project. Instead, a NoSQL database will be selected, which best fits the project's other goals, such as fast communication as well as built in capabilities for deploying listeners for data changes. The server will be the least complicated element of the project, as it only has two major tasks, receiving metrics and passing them along to the database. There are multiple methods to accept the metrics, such as gRPC, REST APIs, or GraphQL, out of which the best method will be selected. As the data exporters are aimed to be implementable in any language and to scrape any type of metric, there is inherently a large number of solutions, which will mostly depend on the communication method chosen while implementing the server. The same applies to the socket, as the communication method between it and clients will be decided upon by examining which best fits the project's vision. Frontend development in general has a vast number of options for implementation ranging from frameworks to libraries, each of which will have advantages and disadvantages to be considered.

3.5 Tools and Techniques

The project will aim to implement all tools and techniques which have been discussed in part of Chapter 2, as they have been researched thoroughly by considering their advantages and disadvantages in relation to this project. For the database, this would imply the use of RethinkDB, as it is specifically built for the real-time web and thus has all functionality this project will require. For communication between exporters and the server, gRPC would be the best choice, as it is the fastest out of all other options and, with a workaround, has the capability to send dynamic data. For communication between the socket and the frontend, web sockets, or more specifically the Socket.IO library, has been researched, therefore the project will aim to implement it. As for the frontend, the VueJS framework is the most likely options, as it has a shallow learning curve. Development will rely on Docker, to provide distributable packages of each component.

3.6 Legal, Social, and Ethical Issues

Inherently, the project, which will be delivered with the artefact, will face no social or ethical issues, as the system only collects data from other machines, such as performance or application metrics.

However, with the aim of this project of being able to collect any type of metric, data exporters which collect sensitive information such as user data could be built and used with the system. To avoid any issues regarding personal data, any future exporters of this nature should adhere to standards defined within the GDPR (Vlahou et al., 2021). Regardless, the project will need to consider and prevent potential security risks, to avoid legal complications. In addition, there are multiple ways with which data could be extracted by a bad actor from any system, therefore the project will need to make sure its communication methods are up to par to prevent intrusions, to avoid potential legal issues.

3.7 Summary

A successful implementation of this project will rely on the methodologies and approaches discussed within this chapter to be acted upon effectively. In part of this chapter, multiple project management techniques were discussed and applied to the requirements of this project. Following the proposed planning should result in the timely creation of the project artefact.

4 Design

4.1 Introduction

This chapter will cover all important design decisions that will influence the implementation and development of this project. This includes not only system design, but also data design as well as user interface design. As design is an important element of any project, this chapter will act as a foundation for the rest of this project's development, including any implementation decisions made down the line.

4.2 System Design

4.2.1 Service Oriented Architecture

Service oriented architecture has already been solidified as a key part of this project, therefore this chapter will adhere to the SOA methodologies. Designing around SOA will make the project not only expandable, but it also enables the development process to be more dynamic, as individual components can be designed separately from the rest of the project without having to adhere to certain limitations other components might face. The following sections will cover both the thought process behind the service design and the decisions made on how data should be transferred and managed between interfaces.

4.2.2 Minimizing Dependencies

The caveat of SOA stems from services being atomic, and therefore only being able to complete the task they were designed for. While this carries many advantages in terms of maintainability, stability, and observability, it also limits individual services to their core objective. This can result in microservice structures with a high amount internal dependencies, as the individual services might not have the capabilities to transform data in a way other services can. This project aims to feature a low number of internal dependencies between services. This won't only help maintainability, but it also gives the project the option of being expanded upon. Further expansions could include overhauls to individual services, addition of other services, and removal of services to change the projects core functionality. To expand on the last point, this might imply the removal of the socket and frontend from the project, thus converting it from a fully featured monitoring solution into an ingress engine.

4.2.3 Minimizing Traffic

One of this project's primary goals is to feature real time data collection and visualisation. To achieve this, communications between services will need to be kept to a minimum, both size and frequency wise. While the SOA methodology generally aids with this goal, services will still need to be designed and built around being real time and asynchronous. The following sections will cover the planned design to achieve this.

4.2.4 Exporter Data Format

A key part of this project will be the data exporters, therefore they are the first component to be considered when designing the rest of the services. As has been mentioned, the data exporters should be able to deliver any type of data to the server. To achieve this, a data interchange format which is both dynamic, and supports multiple data types is required. There are a few formats which meet these requirements, most notably JSON, YAML, and XML. All of these formats feature human-readable structures, while still retaining powerful underlying serialisation and deserialization features. This project will use JSON, as when compared to other formats, it features the fastest serialisation as well as deserialization (Hunter, 2019). Using JSON also has the advantage of being the primary language supported by RethinkDB, the database this project aims to implement.

With the data interchange format being decided upon, the exporters still require a standardized format in which the collected data should be structured. Figure 8 serves as an example of this, the "metrics" datapoint being a dictionary allows for multiple different exporters to be appended dynamically.

```
{
        metrics: {
                CPU: {
                        percent: 2.8
                },
                MEM: {
                        mem_free: 486834176,
                        mem_used: 1105821696
                }
        }
}
```
Figure 8: Proposed Structure of a Data Exporter's JSON Formatted Output

As this project aims to be as efficient and lightweight as possible, the demonstrated exporter output only features free and used memory, as opposed to free, used, and total. The operation of calculating total memory can be outsourced to another component of the project, such as the frontend. This would save both time and transfer size for the exporters and the server.

4.2.5 Protocol Buffer Classes

As mentioned in Section 2.3.2, Protocol Buffers are the underlying data format, which serializes data to be transferred over gRPC. Services, remote procedure calls, and data structures, which are called messages, are defined within a proto definition file. This allows for any data structure to be predefined and sent in a serialized format. This project implements Protocol Buffers as the primary data exchange format between data exporters and the receiving server. To facilitate this, two major remote procedure calls are needed, one to register exporters to the server, and one to push metrics and data to the server.

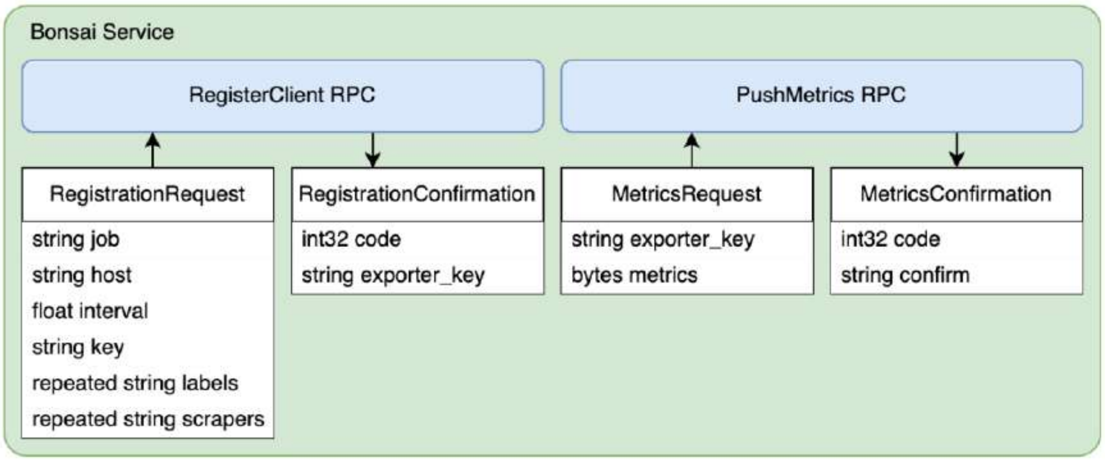


Figure 9: Illustrated Diagram of a Proposed Protocol Buffer Structure

Figure 9 is a visualized representation of what this projects Protocol Buffer definition file might look like. A service containing two procedure calls, each with two messages, one for the requesting entity and one as a return value. As can be seen, most of the exporter's metadata is already sent during the registration step, thus saving bandwidth and time when the exporter starts pushing metrics into the system. Included with the "RegistrationRequest" message is general metadata as well as two repeated strings, functionally comparable to a list, of labels and scrapers. Labels will help users categorize exporters by location, type, and other factors. Scrapers will aid in building dashboards, as they will used to identify which data is collected on certain clients. As the server still needs a method to associate the data it receives with a certain exporter or host, an exporter key is created in part of the "RegistrationConfirmation" message, which is a unique identifier, created by the server, for the exporter. This key is then used for any further communication the exporter establishes with the server, as can be seen in the "MetricsRequest" message.

4.2.6 Dynamic Protocol Buffers

This project aims to be dynamic in nature, to enable any type of data to be collected and stored in the system. While serialized data usually is defined using a static set of variables, this can be worked around. Taking a look back at Figure 9, the actual metrics within the "MetricsRequest" are sent as bytes. This allows for the data to be serialized as bytes before the protocol buffer serialisation. Although doing this adds a step between both the data being sent from the exporter and the data being written into the database, it enables this project to use protocol buffers, while still retaining its dynamic vision.

4.3 Service Design

4.3.1 Planned Structure

This projects service design will stay mostly true to the Illustration shown in Figure 6, which was introduced in Section 2.4.4. The following sections will go into further depth on how communication between services will be established as well as how services will be structured internally.

4.3.2 Data Exporter

Data exporters will be created with pluggability and expandability in mind. This can be achieved by implementing class inheritance. Figure 10 demonstrates this by having one super class "BonsaiExporter" with multiple child classes. The master class defines variables and functions, which have to be shared by all child classes, while child classes still retain the freedom to have their own configuration variables or helper functions. This allows for multiple exporter classes to be added dynamically. The "BonsaiClient" class is then able to simply have a list of initialized child classes of the main exporter class, from which each metric scraping function can be called.

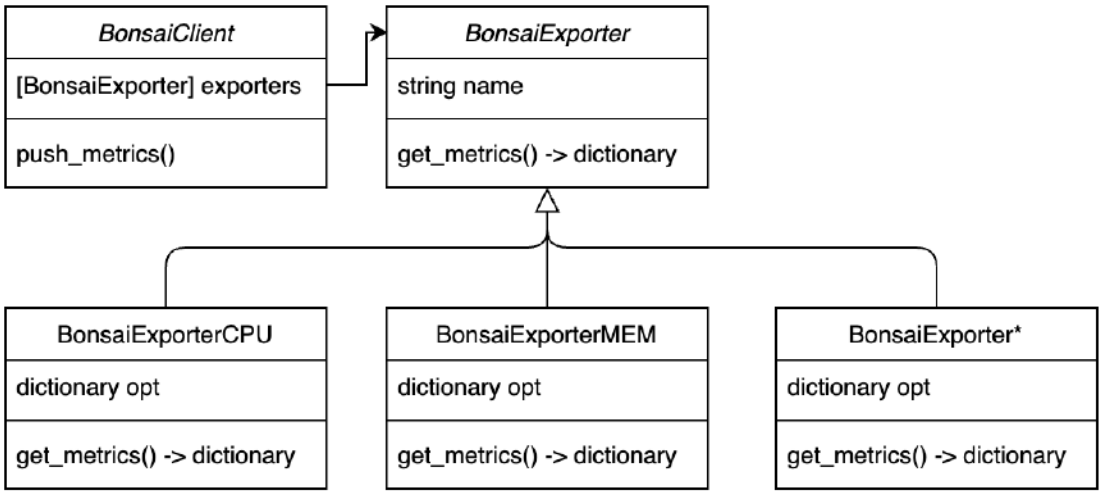


Figure 10: Illustration of a Potential Bonsai Data Exporter Class Diagram

4.3.3 Server

As the server's primary objective will be to receive and process any communication from exporters, its main component will be a gRPC server. Looking back at Figure 9, two remote procedure calls need to be implemented, as well as a middleware, which will need to process incoming data and write it to a database. A high level representation of the actions which need to be completed to achieve this can be seen in Figure 11. A simplified version of an example exporters internal workings is also included to demonstrate the flow of data.

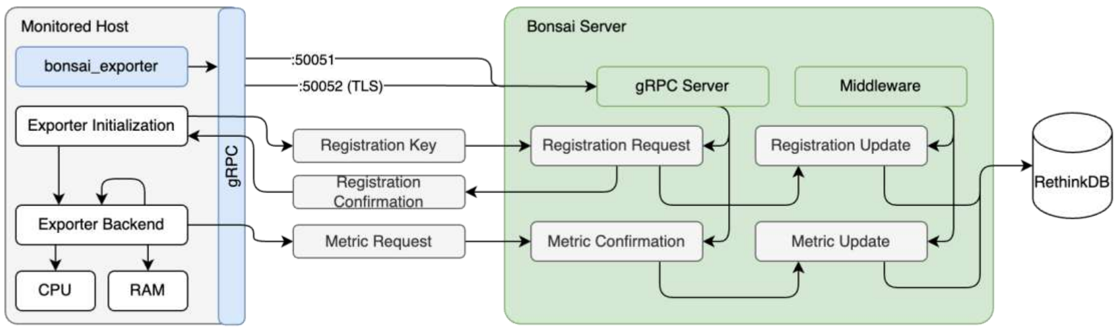


Figure 11: Illustration of the Bonsai Servers Proposed Internal Processes

To briefly explain the diagram, the server sits idle until it receives signs of life from an exporter. The exporter sends over a registration key, which is verified by the server. Should the key be valid, the server first writes the information received from the exporter into the database, then returns an exporter key with the registration confirmation message. Once the exporter receives the key through the confirmation it starts periodically scraping data and sending it to the server using metric requests. The incoming metrics are handled through the server's middleware and written into the database.

4.3.4 Socket

In the spirit of service oriented architecture, the socket will have no communication with the server, to keep internal dependencies to a minimum. Figure 12 clearly demonstrates this, as the only dependency the socket relies on is the database. RethinkDB offers multiple ways to interface with the database, the illustration showing two of the primary options. Regular queries will be used for dashboards, which don't need any real time updates, while a listener pool will be used to deliver real time data to the frontend.

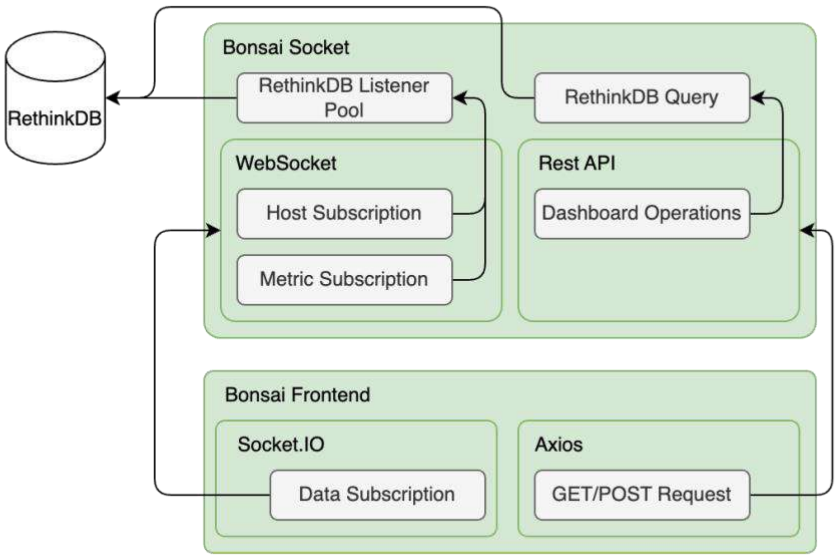


Figure 12: Illustration of the Bonsai Socket's Proposed Communication with the Frontend

4.3.5 Frontend

The frontend, which has already been briefly mentioned, will only depend on the socket as a connection to the rest of the project. The two interface options implemented by the socket, will both be utilized by the frontend. Metrics, or real time data, will be streamed using web sockets, while static information like dashboards will be loaded using a REST API, which will run in parallel with the socket.

4.4 Docker

4.4.1 Docker Images

Docker will be used for the deployment of this project, as its multitude of features and capabilities lend themselves well to the SOA approach. The base of any docker container is an image. Docker images are lightweight, standalone, and executable application packages that contain all requirements the software might need such as a runtime environment, libraries, configurations, and more. A major benefit of packaging software in Docker images is the portability. As an image contains all requirements, it can be deployed on any host with a Docker runtime. This aids both with testing and the final deployment, as Docker containers can easily be migrated or scaled.

4.4.2 Docker Image Size

What takes away from the aforementioned portability is the size of the image. Images are created by combining multiple layers, which are usually defined using steps within a "Dockerfile." Each layer is created by taking snapshots of containers, in which the predefined step has been executed. These layers are then combined to create a final image. Unoptimized Dockerfiles can lead to bloated image sizes. There are multiple factors that play into an images final size. Docker images are usually built using a base image, which can be a primary contributing factor. Taking for example a container image of a compiled language. The base layer for building the application will include all build requirements and compilers. During the initial build, these might be required, but for a final Docker image, the layer containing the build requirements can be discarded, thus reducing the image size. Unnecessary files can also be a factor for oversized images, this can be avoided by copying only the source code into the image instead of the entire repository structure.

4.4.3 Docker Compose

Docker compose is a tool that enables the definition of Docker applications that contain multiple applications using a simple YAML file. Besides containers themselves, Docker compose also offers powerful networking and file management tools, which enables complex application structures to be defined. Within a Docker compose namespace, containers can be linked with each other to enable communication, without having to expose any ports. Databases, for example, can be completely isolated from the outside by only linking them to the application that requires access. During development, where data is not required to be persistent, Docker compose is able to manage volumes, which act as virtual filesystems mounted within containers. This is ideal for development, as data created within containers gets reset once the container is redeployed. The database contents can therefore be kept in a volume even when redeploying the entire application. Docker compose also enables developers to easily monitor resource usage of individual containers as well as access to all logs of all or individual containers, thus enabling monitoring of interactions between containers. Thanks to Docker compose being able to manage the builds of multiple containers, entire applications can be built with just one command. This makes it easy to share and distribute even complex applications, as dependencies are installed during the predefined build process. This also means that applications can be fully built and deployed with just two commands. One to build the components and one to start all containers. This also makes the move to a production environment simple.

4.4.4 Docker Compose Structure

This project will leverage Docker composes capabilities both for testing and deployment. Figure 13 serves as a visual aid for what this project's Docker compose file will look like. The green squares being containers within a namespace. Blue bidirectional arrows illustrate internal links between containers. Black arrows show ports which are exposed to the outside of the containers and namespace. Purple arrows display volume mounts. To further demonstrate this project's dedication to SOA and minimal internal dependencies, the connections between containers were kept to an absolute minimum while still retaining complete functionality.

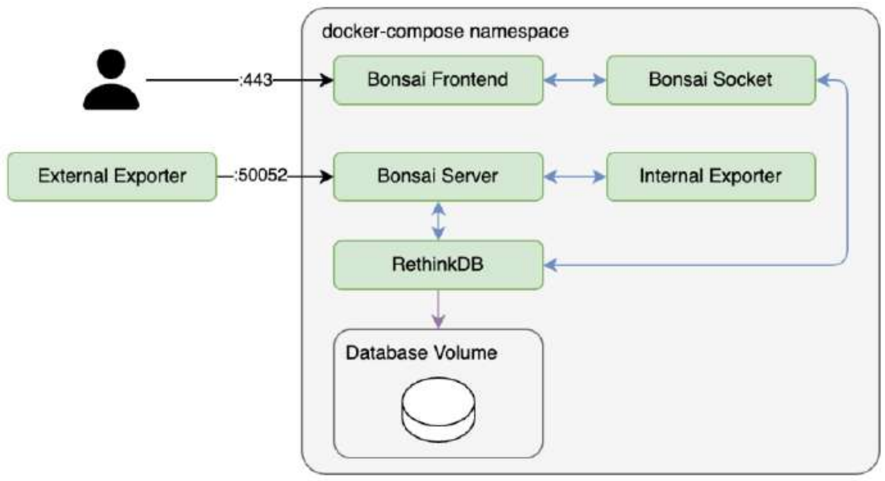


Figure 13: Illustration of the Proposed Docker Compose File Structure

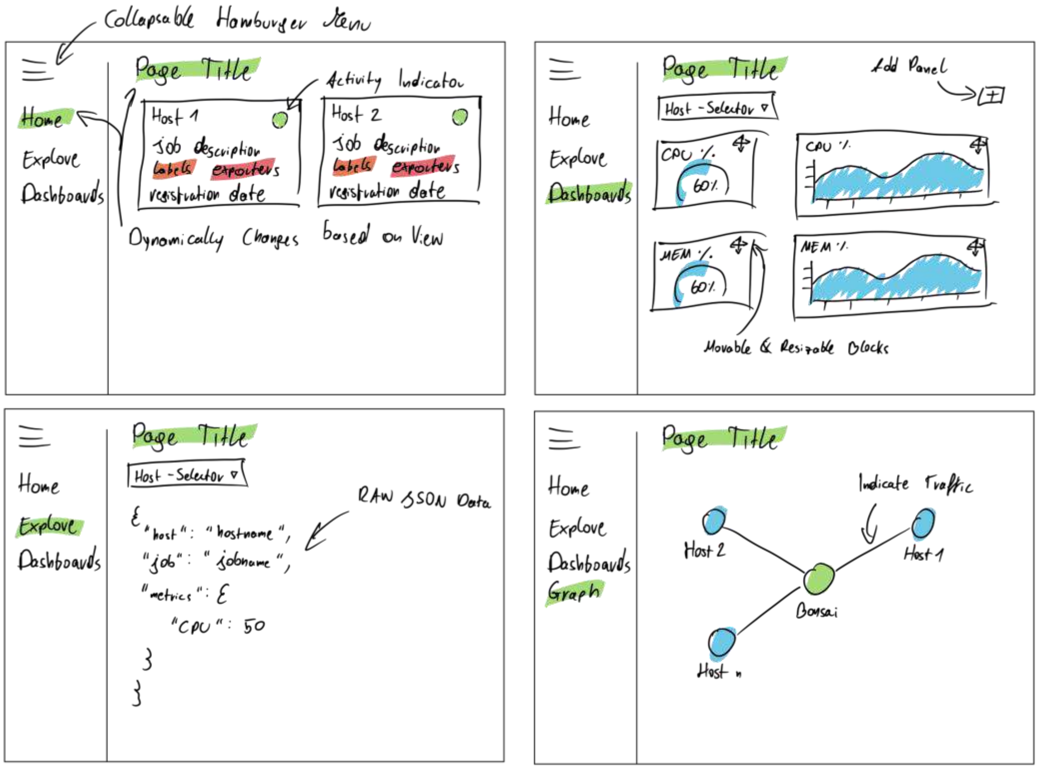
4.5 User Interface Design

4.5.1 Views

This projects frontend will provide users with a way to interact with the data flowing through the system. In VueJS the DOM, is displayed within a view, which can be dynamically interchanged. This project aims to implement four views, a home page, an explore tab, a dashboard view, and a graph view. The home page will give a simple overview of all hosts, which are currently registered to the server. The explore view will be a tool to get the raw data a specific exporter last sent to the server. The dashboard view will be most complex, as it should be able to provide users with a way to dynamically create custom dashboards. This will be achieved by implementing resizable, movable, and customizable panels, which contain charts or graphs to visually display metrics. The inspiration for this approach has been drawn from Grafana, which features a similar system to create its dashboards, as can be seen in Figure 1. Finally, the graph view will be a visual representation of the exporters and bonsai server, with some sort of activity indicator to show when exporters have pushed new metrics.

4.5.2 Wireframes

As the user interface will be the first interaction most people will have with this project, it will need to make a good first impression. To achieve this, rough sketches were created to guide the actual implementation of the frontend. These sketches, which can be seen in Figure 14, serve as the wireframes for this project's frontend. Wireframing is the process of creating visual representations of a web applications interface, content, layout, and functionality. Starting with wireframes before development is essential, as it gives developers a basic layout to start with, thus saving time. Going over the presented wireframes, the four views all have a planned layout and functionality. The frontend will be built as a single-page application, meaning that the application loads new content dynamically by rewriting the current page, instead of loading an entirely new page. Therefore, all views will have a collapsible side menu, which serves as a navigation tool through the separate views. The home view features simple blocks containing every host's metadata, such as hostname, labels, exporters, and registration date. The explore panel will display a drop down menu, in which any available host can be selected. The selected hosts raw metrics will be displayed below. The dashboard view will also feature a host selector, to control which hosts metrics are fed into the individual charts. As can be seen in the wireframe, the charts will be made to be movable and resizable. The graph view will be a node graph that includes the server and every registered exporter as nodes. This will give users a way to visualize the monitoring infrastructure.



```
{
 "host": "hostname",
 "job": "jobname",
 "metrics": {
    "CPU": 50
 }
}
```

Figure 14: Wireframes of the four Planned Frontend Views

4.5.3 Colour Scheme

Alongside presentability, a consistent colour scheme is essential for web applications, as it creates a cohesive visual identity. While helping establish a layout and improving user experience, colour is a tool to convey emotion (Cyr et al., 2010). This project will use the "Nord" colour pallet, which was created for clean and uncluttered design, to achieve optimal focus and readability for UI components (Greb, 2016).

4.6 Summary

This chapter has covered the multitude of different design approaches, both technical and graphical, that will be used during development of this project. Following the methodologies laid out within this chapter will ensure a satisfactory implementation of the system, able to fulfil all requirements without making any compromises.

5 Implementation

5.1 Introduction

Due to this project having many components, each fulfilling a different role within the system, this chapter aims to introduce each of the elements in the context of their implementation, primarily by using code snippets. Each component of the project will be covered to iterate on their main functionalities, and how they were implemented.

5.2 Protocol Buffers

5.2.1 Introduction

The capabilities of protocol buffers have already been discussed within this report in Section 2.3.2, this section aims to cover how definition files are used within a project. Once a definition file has been created, `protoc`, the protocol buffer compiler, is used to convert services and messages into code. This is done through the process of "code generation," which returns source code files in the language specified during the compilation process. Definition files can be compiled into multiple languages due to `protoc` supporting plugins, either official or from the community.

5.2.2 Bonsai Proto File

The definition file used for Bonsai, a section of which is shown in Listing 1, is used both to establish communication standards, and to create objects and remote procedure calls to be used by this project's services. As can be seen in the listing, the remote procedure calls within the `BonsaiService` are defined to only accept and return certain messages. The variable types within messages are statically typed.

```
package bonsai;

service BonsaiService {
  rpc RegisterClient (RegistrationRequest) returns (RegistrationConfirmation) {}
  rpc PushMetrics (MetricsRequest) returns (MetricsConfirmation) {}
}

message MetricsRequest {
  string exporter_key = 1;
  bytes metrics = 2;
}
```

Listing 1: [bonsai.proto] Snippet of the Bonsai Proto Definition File

5.3 Data Exporter

5.3.1 Introduction

As laid out in Section 4.3.2, the data exporter this project was submitted with was built to be pluggable. In this context, pluggable means that additional exporter classes can be added simply through the configuration of the exporter.

5.3.2 Configuration

Exporter configurations are used to initialize the exporter with the information it requires to operate. This includes the exporter's hostname, the receiving server, the rate, or interval, with which the exporter aims to send data, the exporters labels, and the exporter classes to be used. An example of a complete configuration can be seen in Listing 2, which sets all of the mentioned variables.

```
hostname: report-exporter
bonsai_server: server:50051
rate: 1
labels:
 - python
 - report
exporters:
 BonsaiExporterCPU:
   options:
     "individual_cores": True
     "core_count": True
     "core_count_logical": True
 BonsaiExporterMEM:
   options:
     "include_swap": False
     "detailed": True
```

Listing 2: [report_config.yml] An Example of a Complete Bonsai Exporter Configuration

5.3.3 Exporter Classes

As this project aims to be as dynamic as possible, exporter classes were implemented in that sense. As introduced in Section 4.3.2, exporter classes should inherit from a base class. The base class is purposely kept as minimal as possible, to enable further expansion for potentially more complex future exporters. There are only two requirements for any exporter class, a variable, which defines its name, and a `get_metrics()` function, which returns a dictionary of the collected data. The base class is not functional, as a `NotImplementedError` is thrown when trying to call its scraping function, thus making it an abstract class.

5.3.4 Pluggable Exporters

While exporters can be written in any language due to the communication being implemented using gRPC, the exporter submitted with this project's artefact was written in Python. Python, being an interpreted language, has the advantage of being both easy to work with and very dynamic in nature. Python therefore is able to import classes dynamically during its runtime. Seeing Listing 3, a snippet of the BonsaiConfigLoader class, additional exporters are imported by first reading the exporter class names from the configuration and then imported using `getattr(__import__("ExporterClass”))`. Due to the exporters being loaded in dynamically, additional exporter classes can be written and implemented without any change to the exporters code.

```
class BonsaiConfigLoader:
   def __init__(self, config_name='config.yaml'):
      with open(config_name, 'r') as file:
         self.config = yaml.load(file, Loader=yaml.Loader)

   def __repr__(self):
      return json.dumps(self.config, indent=2)

   def create(self):
      for exporter in self.config['exporters']:
         exp_class = getattr(getattr(__import__("exporters." + exporter), exporter),
exporter)
         self.config['exporters'][exporter]['class'] = exp_class
```

Listing 3: [BonsaiConfigLoader.py] Snippet of the BonsaiConfigLoader Class

The BonsaiConfigLoader class then creates a final BonsaiClient class, which periodically scrapes data and sends it to the server using all information and exporters defined within the configuration.

5.3.5 Entrypoint Script

To make the exporter even more customizable, even after being packaged in a Docker image, a start script was created to install packages and mount any additional exporter classes. The entrypoint script first checks if the path to custom exporter classes exists and copies them to the other exporters location. Should it be defined, the $PIP_INSTALL environment variable, a comma separated list of any dependencies, is iterated over and the packages are installed. Finally, the script executes the main file and sets the process id. This approach allows for any custom monitoring endpoints to be added to the pre-built Docker image without any major effort. In practice, running an exporter with custom classes can be done with the following command:

```
docker run -e PIP_INSTALL=docker,docker-py -v "./testing/exporter:/opt/custom_exporters"
konstfish/bonsai_exporter
```

The command starts a Docker container from the exporter base image and sets the environment variable responsible for installing additional Python packages, as well as mounting a directory containing custom exporter classes in the container.

5.3.6 Client Class

As the BonsaiClient class code is too long to fit within this report, a diagram of its inner workings can be seen in Figure 15.

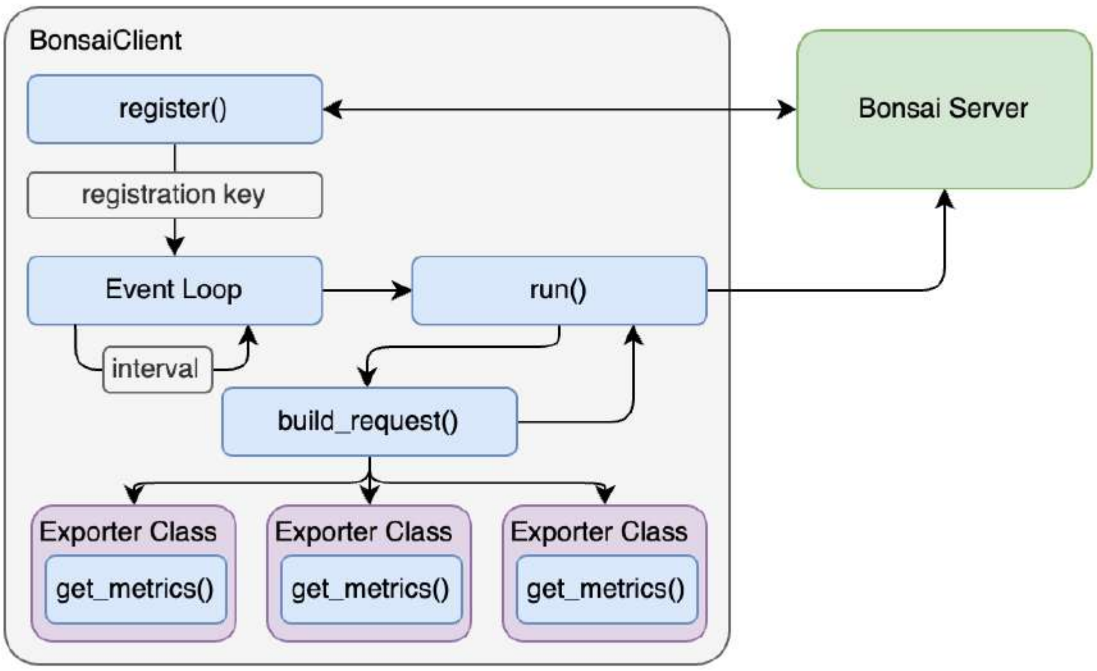


Figure 15: The BonsaiClient Class Implementation Illustrated as a Flowchart

Going over the illustration, once the BonsaiClient has been initialized by the BonsaiConfigLoader, its first act is to register itself with the Bonsai server. If the registration is successful, the exporter receives a key, with which further communication is authorized. Once registration is complete, an event loop starts in which the run() function is called in the interval defined in the configuration. The run function first calls a separate function, build_request() which calls the get_metrics() function of each loaded exporter class and returns a complete dictionary of each. The complete dictionary along with the key is then sent over to the Bonsai server.

5.3.7 Server Communication

Listing 4 is a snippet of the BonsaiClient class, with two of the major functions. build_request(), which creates a protocol buffer MetricsRequest object with the collected metrics and key.

send_simplified(), which in the interest of brevity, has been shortened to only include the code which creates a channel to the server and sends the metrics request.

```
class BonsaiClient:
   def build_request(self):
      for exporter in self.exporters:
         data[exporter.name] = exporter.get_metrics()
      # create protobuf object
      return metric_req

   def send_simplified(self):
      metrics = self.build_request()

      with grpc.insecure_channel(self.bonsai_server) as cnl:
            stub = bonsai_pb2_grpc.BonsaiServiceStub(cnl)
            response = stub.PushMetrics(metrics)
```

Listing 4: [BonsaiClient.py] Snippet of the BonsaiClient Class to Demonstrate Request Construction

5.4 Server

5.4.1 Introduction

To ensure speed, the server implementation has been kept very minimal, only serving as a layer between exporters and the database. There are two major elements to the server's structure, the implementation of the gRPC services, first mentioned in Figure 9, and a handler for communications with the database.

5.4.2 gRPC Service Implementation

As the Bonsai server's primary objective is to act as a receiver for any and all requests from exporters, it implements the generated code from the protocol buffer definition file. As can be seen in Listing 5, this is done by creating a service class that inherits from the generated code. The class then implements the remote procedure calls with the same arguments and return values specified within the definition file. This class can then be added to a gRPC server, which handles requests from clients.

```
import bonsai_pb2
import bonsai_pb2_grpc

class BonsaiServer(bonsai_pb2_grpc.BonsaiServiceServicer):
   def __init__(self, key=None):
      self.key = key

   async def RegisterClient(self, request: bonsai_pb2.RegistrationRequest) ->
bonsai_pb2.RegistrationConfirmation:
      logger.info('Registration request from host %s!' % request.host)
```

Listing 5: [BonsaiServer.py] Snippet of the BonsaiServer Class

5.4.3 Database Controller

In favour of a connection pool, the server connection handler class has been implemented to open new connections for each set of operations, as this enables the use of the Python with statement. The with statement is used to handle objects that need to be cleaned up or released when no longer needed, such as database connections. This is implemented by creating a class with the __enter__ and __exit__ functions, which serve as the constructor and destructor respectively. Listing 6 is a section of the database controller source code, which demonstrates how the database connection is implemented. The enter function establishes and returns a connection object, while the exit function closes the connection. The create_table() function, which has been simplified for demonstration purposes, then creates a connection object using the with statement, runs an operation using it, after which the connection is cleaned up automatically.

```
class RethinkServerConnection():
  def __init__(self, rethink_server: RethinkServer):
    self.rethink_server = rethink_server
    self.conn = 0

  def __enter__(self):
    self.conn = self.rethink_server.r.connect(  self.rethink_server.rethink_server,
                                      self.rethink_server.rethink_port,
                                      db=self.rethink_server.rethink_database)
    return self.conn

  def __exit__(self, *args, **kwargs):
    self.conn.close()

def create_table(table_name, database_name, rethink):
  with RethinkServerConnection(rethink) as conn:
    rethink.r.db(database_name).table_create(table_name).run(conn)
```

Listing 6: [RethinkServerConnection.py] The RethinkServerConnection Class

5.4.4 Middleware

Now that both the gRPC server and database connection are established, the corresponding middleware can be covered. Listing 7, which is a simplified version of the PushMetrics remote procedure call implementation, serves as an example of the process metrics go through once they arrive at the server. First a JSON object is created using the data from the serialized protocol buffer message. They are decoded using Python's built in JSON library and a timestamp is appended to the complete metric object. A connection to the Rethink database is established, and the premade JSON is inserted in place of the previous entry. Once complete, a MetricsConfirmation message is created and sent back to the client. The complete function features more thorough error handling and sends back different MetricConfirmation messages based on how the server processes the message. Error codes were made to be similar to HTTP status codes, the server returns 200 on success, 401, if the host is unauthorized and 500, should the server face any issues.

```
async def PushMetrics(self, request: bonsai_pb2.MetricsRequest,
            context: grpc.aio.ServicerContext) -> bonsai_pb2.MetricsConfirmation:
    rjson = {
        'id': request.exporter_key,
        'metrics': json.loads(request.metrics.decode('utf-8')),
        'date': str(datetime.now())
    }

    # write metrics into rethinkdb
    with RethinkServerConnection(rethink_server=rethink) as conn:
        out = rethink.r.table("metrics").insert(rjson, conflict="update").run(conn)
        if(out['replaced'] == 1):
            logger.info('Replaced metrics for host %s' % request.exporter_key)
        else:
            logger.info('Recieved initial metrics for host %s' % request.exporter_key)

    # return MetricConfirmation
    return bonsai_pb2.MetricsConfirmation(code=200, confirm="success")
```

Listing 7: [BonsaiServer.py] Simplified Snippet of the BonsaiServer Class PushMetrics Function

5.4.5 Reflection and Health Monitoring

Server reflection is a technique which allows for runtime analysis of a server's state and functionalities. In this sense, gRPC reflection allows clients to query a gRPC server to gather information on available remote procedure calls and the corresponding available messages. It was implemented in this project to aid with further development and debugging by using the add-on library provided by the gRPC authors (The gRPC Authors, 2023b). Furthermore, the gRPC health checking protocol was implemented. This service allows clients to query whether a remote procedure call is currently able to process requests. This can then be used for systems like load balancers to determine which host should handle the incoming request.

5.4.6 TLS Implementation

TLS, or Transport Layer Security, is a protocol which enables secure communication between two hosts using certificates. An illustration of how TLS achieves this can be seen in Figure 16. TLS comes with several benefits, such as enhanced security in transit and protection against potential attacks. This project opted to include TLS as a method to authenticate the server, as it is already supported by gRPC (The gRPC Authors, 2023a). Once the certificates have been created, implementing them only requires the change of the `insecure_channel` gRPC directive to the `secure_channel` directive, with the certificate information passed as an argument.

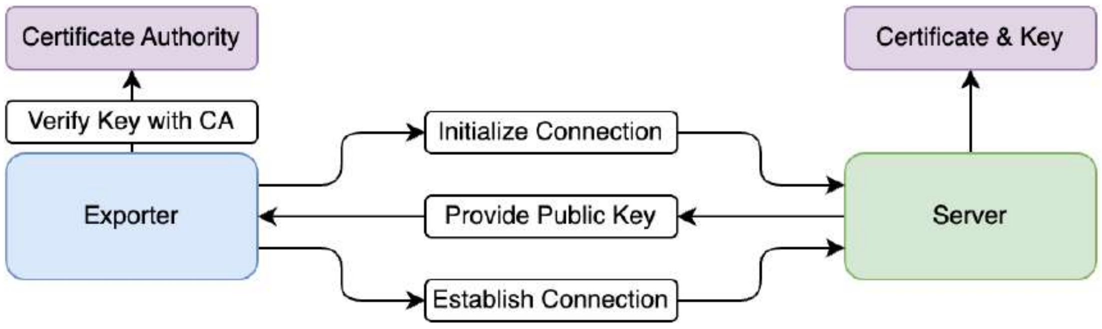


Figure 16: Illustration of the TLS Connection Process

5.5 Socket

5.5.1 Introduction

The socket implements two major elements, a web socket, and a REST API. The web socket is used to transfer metrics to the frontend, while the REST API is used for the transfer of more static data, like dashboards. The socket was written in NodeJS, as there are a multitude of existing libraries for web related applications.

5.5.2 Database Connection Pool

The socket implements a connection pool to interface with the Rethink database. This is done using the "rethinkdbdash" library, a NodeJS driver for RethinkDB (Michel, 2014/2023). This library was used in favour of the official RethinkDB API, as it features more advanced features such as a built in connection pool capability and simpler function calls. A RethinkDB driver is created by initializing the rethinkdbdash library with information on the server or multiple servers. The resulting class is then exported by the module, which can then be used in any part of the socket application, by importing the module.

5.5.3 Socket.IO

To implement web sockets, the Socket.IO library was used, which has already been explored in Section 2.3.4. Going through the snippet shown in Listing 8, setting up an instance of Socket.IO is only a matter of a few lines of code. First, a HTTP server, in this projects case "express," is required as a base for the Socket.IO server. Once both the HTTP and Socket.IO servers are defined, the Socket.IO server is initialized with the `io.sockets.on()` function. Incoming requests are handled using this function, these requests consist of simple messages in a JSON format. The "type" key indicates what is required of the socket, while other keys can be used to transfer additional information, like filters. There are two response methodologies shown in the snippet. The first, get_hosts, simply returns a list of all available hosts, retrieved from the database. The second, update_listener, starts a subscription which actively sends new metrics to the client socket, which initialized the listener. This route can be continuously called to recreate the listener with new filters, in this example, filters are applied by label.

```
var app = express();
var server = require('http').Server(app);

var io = require('socket.io')(server, { path: '/ws' });

io.sockets.on("connection", function(socket){
    console.log("connection -", socket.id)

    socket.on("message", (data) => {
        const packet = JSON.parse(data);

        if(packet.type == 'get_hosts'){
            // retrieve hosts
            dbController.getHosts(function(res){
                socketController.brodcastMessage("host_list", res, socket)
            })
        }

        // listeners
        else if(packet.type == 'update_listener')
        {
            var labels = packet.content[0];
            dbController.getMetricsByLabelListener(socket, labels);
        }
    });
})
```

Listing 8: [index.js] Snippet of the Bonsai Socket Index File

5.5.4 Socket Routes

As has been introduced in the previous Section, the socket features multiple routes, all having their own purpose. Covering the static requests first, three of which are implemented by the socket. get_labels, which returns a list of all labels used by exporters, get_hosts, which returns a list of all host identifiers currently pushing metrics into the system, get_hostnames, which returns a list of all hostnames. Continuing with the listener routes, which are key to the functionality of this project. update_listener is used to create a listener on the database, which continually sends any and all changes to the metrics, from hosts which have a specific label. update_listener_metric_host, which filters by host identifier, instead of labels. And finally, update_listener_host, which is used to subscribe to all host metric changes.

5.5.5 Listener Implementation

This section aims to cover how listeners are implemented, taking the update_listener route as an example. Two functions are used to achieve this, one to create a database cursor, and one to iterate the cursor and one to transfer new rows to the client, which created the listener. The first function getMetricsByLabelListener(), takes two arguments. The client socket, which created the connection and a list of labels to filter metrics by. A cursor is created using the set of labels, and then passed on to the second function, pushMetricChanges(). Using the .changes() function provided by the

RethinkDB library, the cursor acts like a stream, supplying new rows as soon as any change is made to the database. Therefore, the push function can simply iterate over the cursor endlessly and send any and all changes to the client. On every iteration, a check is made to see if the client still has the socket connection open. Should this check fail the cursor is closed and the function exits.

5.5.6 REST API

To assist the web socket, a simple REST API was implemented. REST API, standing for Representational State Transfer Application Programming Interface, is an architectural style created for web services. It can be implemented using HTTP methods, like GET, POST, or PUT, to perform operations on resources, which are identified by URL. The REST API implemented in part of the socket is used to manage data related to dashboards. As the socket already features an express server, routes can simply be created and appended to the existing server. Listing 9, is an example of how routes are defined as functions and then exported to be implemented in the index file. A router object is created and functions for each required route and method are appended. Each function receives two variables as arguments. The first argument `req` contains details about the client and any headers or body content, which might have been sent along with the request. The second argument `res` is used to send a response to the client.

```
var controller = require('../controllers/dashboardController')
var router = express.Router();

// adds a dashboard to the DB
router.post('/add', (req, res) => {
 try{
   console.log(req.body)

   controller.addDashboard({name: "Untitled Dashboard", layout: []}, (err, task) => {
    if(err) throw err;
    console.log(task);
    res.status(200).json({"status": 200, "task": task});
   })
 }catch(e){
   console.log(e)
   res.status(400).json({success: false, msg: 'GENERAL'})
 }
})

module.exports = router;
```

Listing 9: [dashboardRoutes.js] Snippet of the Bonsai Socket Dashboard Route Definitions

These routes are then appended to the existing express server by importing them in the index file using `var routesDashboard = require('./api/routes/dashboardRoutes')` and then adding the imported router object to the server `app.use('/api/dashboards', routesDashboard)`. For the implementation in the socket component, two separate routes were created, one for dashboard management, and one for administration tasks, such as health monitoring.

5.5.7 REST Controller

To keep the router functions simple, a controller class is used to handle any interaction with the database. To enable communication with the database, the rethink module introduced in Section 5.5.2 is included. The functions in the controller module take two arguments, one used to exchange information and one as a callback function, which is called once the database operation is completed. This style of programming is required when creating sequential operations in JavaScript, as it is a non-blocking language, meaning that upon execution, operations are started in parallel without regard if the previous operation is complete. In this context, the information argument is used to either pass information, which is then written into the database, or to specify search parameters. Once the database operation is complete, the callback function is called, which then causes the REST API to send a response to the client.

5.6 Frontend

5.6.1 Introduction

The final system component discussed in this chapter is the frontend. The frontends aim was to provide a visual interface for the metrics flowing through the system. Therefore, it is also the last destination metrics reach on the journey from an exporter through the server, database, and socket. As the functionality of VueJS has already been introduced in part of this report's Chapter 2, it was used to implement the frontend.

5.6.2 Vue Router

One of the core functionalities that comes with VueJS is the built in router, which provides a way to navigate through the different views of a VueJS application. The Vue router is entirely client-side, which comes with a number of benefits such as faster page transitions and better scalability, as the server won't need to serve content for every view change. Each route controlled by the router is associated with a Vue component, or view. Once the webpage is navigated to a route, the corresponding view is rendered into the DOM. As can be seen in Figure 17, an example Vue router web application, some elements are static, such as the sidebar, which acts as the navigator. Once a new route is accessed the existing views content is replaced by the new view. In the example below, "View 1" is currently loaded in, navigating to "View 2" would unload the first views content and replace it with the new view, leaving every other element of the application the same during the process.

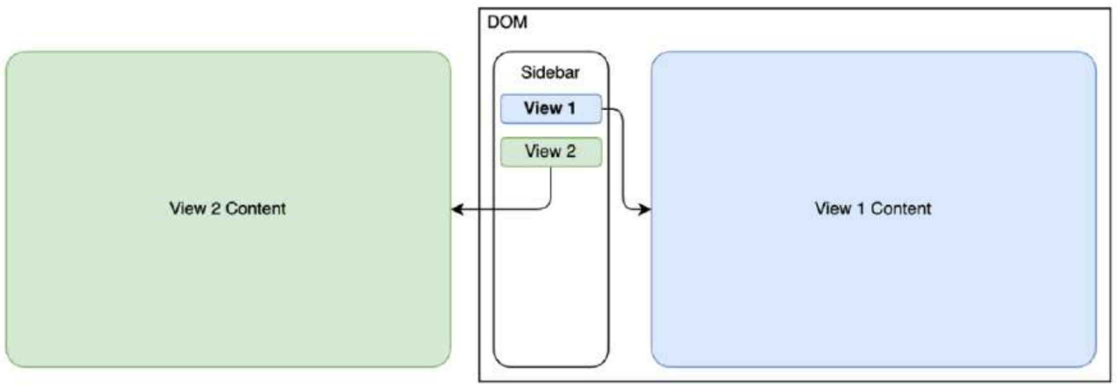


Figure 17: Illustration of a Web Application Implementing the Vue Router

5.6.3 Socket Communication

To achieve communication with the socket, the Socket.IO client library is implemented (Socket.IO, 2023). Listing 10 is a simplified version of the home view's script section, demonstrating how the socket connection is established and used. Initially, the socket object is defined in the ViewModel layer of the Vue application using the server address and path to the socket. Once the component has been created, the socket is opened. The on directive is then used to define what should happen once the socket client receives communication from the server. In this case, once a host update is received, the data is appended to the hosts dictionary. To initialize communication the send directive is used to register the client with the socket server. To ensure no unnecessary traffic flows through the system, socket connections are closed once the view is unmounted or closed.

```
export default {
   data() {
         hosts: {},
         socket: io(this.socket_io_server, {path: "/ws"}),
   },
   created() {
    this.socket.open()
    this.socket.on("hosts_general_update", (row) => { this.hosts[row.id] = row });
    this.socket.send(JSON.stringify({ type: "update_listener_host" }));
   },

   unmounted() {
    this.socket.close()
   },
}
```

Listing 10: [HomeView.vue] Simplified Snippet of the Script Section of the Home View

5.6.4 Home View

As laid out in Section 4.5.1, four primary views were created in part of implementing this projects frontend. Figure 18 shows the frontends landing page, demonstrating that the decisions made during the design process were adhered to. Going over the home page, the navigation sidebar clearly lists all available views using corresponding iconography to improve accessibility. To reconfirm what view is currently selected, a header bar is used. The home view's primary objective is to give an overview of what hosts are currently registered to the system, allowing users to quickly identify any issues. Each cell represents an individual exporter, containing the hostname, exporter name, a list of labels in orange, a list of exporters in red, and the time since registration. Additionally, two indicators are placed on the side of each cell, the top indicator flashing green each time new data is received from the exporter and the bottom indicator signalling exporter health based on matching the frequency of received data with the value set by the exporter.

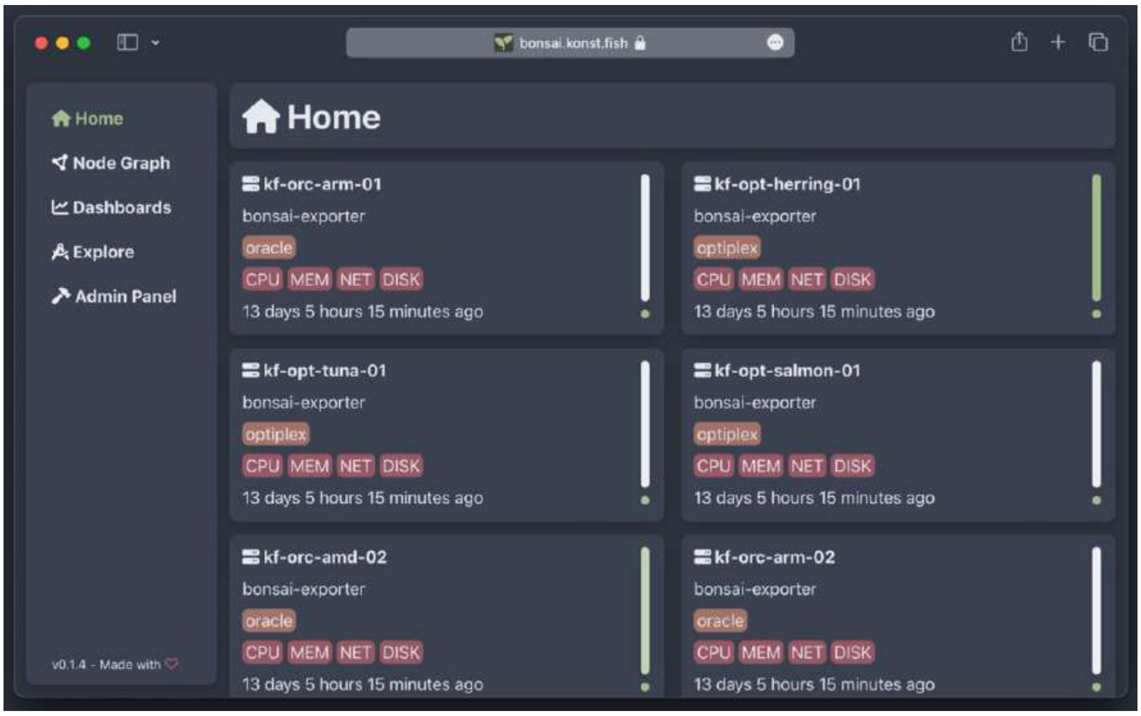


Figure 18: Bonsai Frontend Home View

The cells are created dynamically by implementing the v-for directive, as can be seen in Listing 11. This directive allows for list objects to be iterated over, with an element being created in the DOM for every entry contained within the object.

```
<template>
   <div class="node" v-for="host in this.hosts" v-bind:key="host">
     <div class="hostname">{{ host.host }}</div>
     <div class="job">{{host.job}}</div>
     <div class="labels">
       <span class="label" v-for="label in host.labels">{{ label }}</span>
     </div>
     < time-since :date="new Date(host.registration_date)" />
     <div class="update-stripe" :id="host.id"></div>
     <div class="live-circle" :id="'live-circle-'+hosts_status[host.id].status"></div>
   </div>
</template>
```

Listing 11: [HomeView.vue] Simplified Snippet of the Template Section of the Home View

5.6.5 Dashboarding View

The dashboarding view had the biggest scope when compared to the other frontend views. Two major libraries were implemented to achieve the dashboard implementation. To create a dynamic canvas of cells which can be resized and repositioned dynamically, the vue3-grid-layout library was used (JBay Solutions, 2021). To fill these cells with content, the ApexCharts library was used, an open-source library of modern chart designs (ApexCharts, 2023). An example dashboard in the finished implementation of the dashboarding view can be seen in Figure 19.

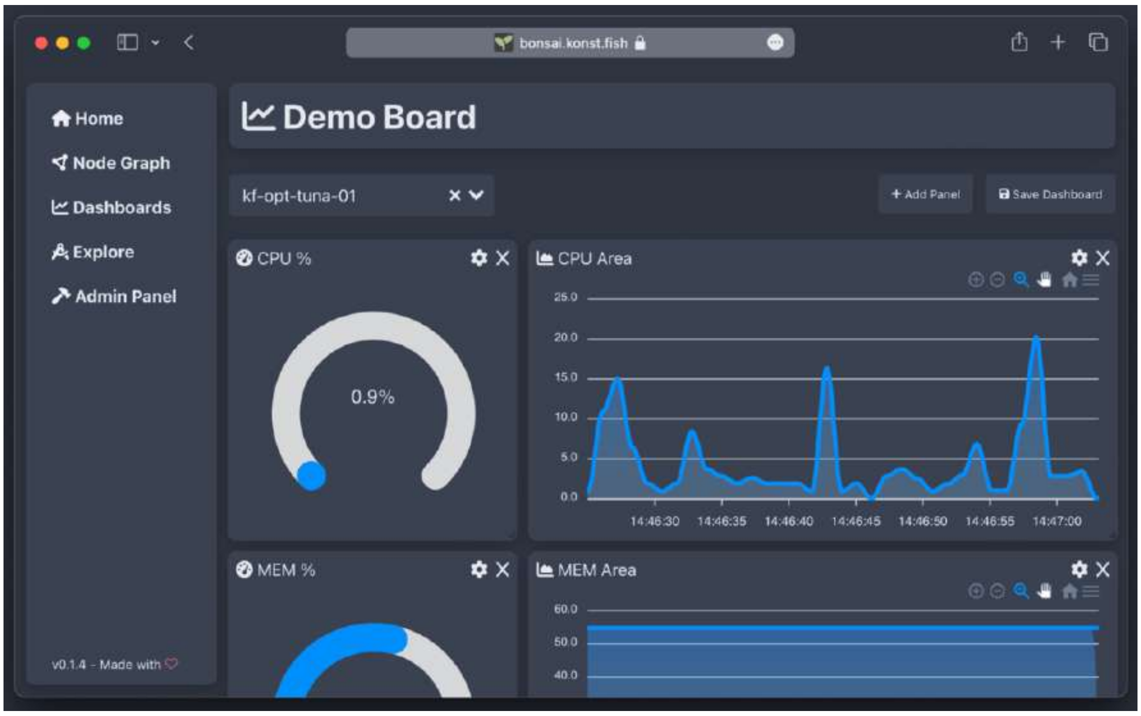


Figure 19: Bonsai Frontend Dashboard View

The dashboard view initially has no content, other than a dropdown selector, with which a host can be selected to subscribe to its metric stream, a button to add a panel, and a button to save any changes made to the dashboard. The add panel button causes a pop up menu to be shown in which a title, the metric point to be passed to the chart, and a chart type can be selected. The metric points are loaded dynamically, depending on which host has been selected in the dropdown menu. Once a panel has been created it can be repositioned and resized in accordance with other panels. The dashboarding view currently supports four chart types, gauges, multi-gauges, area charts and bar charts. The charts are loaded dynamically through Vue components, meaning that future addition of charts only requires the creation of an additional component. If the data point defined with the panel is found in the metrics from the subscribed host, it is passed through to the component, updating the chart each time new metrics are received. As can also be seen in the dashboard view example, the dashboarding view can be used to observe metrics over longer periods of time, as a limited number of points are saved in the client's web browser to be charted out in area charts.

Dashboards are saved as an array of dictionaries, each containing information on a cell's position, size, chart, and metric point. Figure 20 is an example of how a dashboard is returned when requested from the REST API. The dashboard shown in the example contains two cells, a gauge displaying the `CPU.percent` metric and an area chart displaying the `MEM.percent` metric. The specific dashboard is accessed using an URL parameter, from which point the frontend sends a request to the socket to get the layout information. This layout information is dynamically loaded each time a specific dashboard is accessed.

```json
{
   "id": "4d9df8f0-221c-41b1-bfea-d0355d52ab94",
   "name": "asdf",
   "layout": [
      {
         "h": 7, "w": 4,
         "x": 0, "y": 0,
         "metric": "CPU.percent",
         "metric_type": "single",
         "name": "CPU%",
         "type": "singlegauge"
      },
      {
         "h": 7, "w": 12,
         "x": 0, "y": 7,
         "metric": "MEM.percent",
         "metric_type": "multiple",
         "name": "MEM Area",
         "type": "areachart"
      }
   ]
}
```

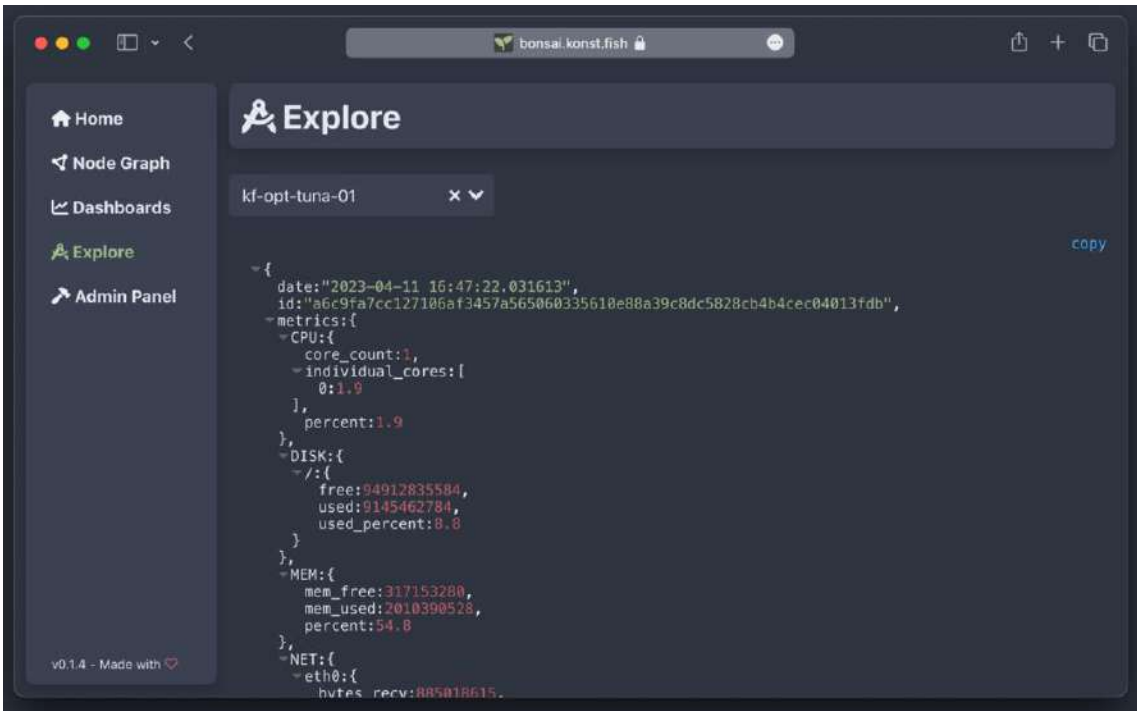
Figure 20: Example Dashboard Data Structure

5.6.6 Node Graph View

To provide a visual representation of the monitored infrastructure, the node graph view was implemented. Using the v-network-graph library, a node graph is dynamically created from the information drawn from the socket (dash14.ack, 2023). The graph nodes, which are exporters, are connected to their corresponding labels to enable the quick recognition of which hosts belong to what group. An example of the graph view can be seen in Figure 22.

5.6.7 Explore View

Implementation wise, the explore view can be considered the simplest. Similar to the dashboarding view it features a dropdown menu with which a host can be selected. After a host is selected a subscription to the hosts metric stream is created. The data received from this stream is then displayed and actively updated, as it is received from the socket. The explore view primarily serves as an alternative to the more visual dashboarding view, allowing users to more intricately view the metrics exported by each host.



```
{
  date:"2023-04-11 16:47:22.031613",
  id:"a6c9fa7cc127106af3457a565060335610e88a39c8dc5828cb4b4cec04013fdb",
  metrics:{
    CPU:{
      core_count:1,
      individual_cores:[
        0:1.9
      ],
      percent:1.9
    },
    DISK:{
      /:{
        free:94912835584,
        used:9145462784,
        used_percent:8.8
      }
    },
    MEM:{
      mem_free:317153280,
      mem_used:2010390528,
      percent:54.8
    },
    NET:{
      eth0:{
        bytes_recv:885018615,
```

Figure 21: Bonsai Frontend Explore View

5.6.8 Reverse Proxy

CORS, standing for Cross-Origin Resource Sharing, is a security restriction implemented by web browsers to prevent websites from making requests to any other websites. Even a change in port number can trigger CORS to block a request, therefore a reverse proxy was implemented to circumvent this when making requests to either the web socket or REST API implemented by the socket. As the frontends Docker container builds upon NGINX, a capable web and reverse proxy server, it was configured to serve content from the socket, as can be seen in Listing 12. This also results in the entire project only requiring two ports to be opened on a server, one for the server and one for the frontend, or reverse proxy.

```
http {
  server {
    listen 3000;

    root     /usr/share/nginx/html;
    # static content
    location / {
      try_files $uri $uri/ /index.html;
    }
   # web socket
    location ^~ /ws {
      proxy_pass http://socket:9000/ws;
    }
   # api
    location /api {
      proxy_pass http://socket:9000;
    }
  }
}
```

Listing 12: [nginx.conf] Simplified NGINX Config Implemented by the Frontend Docker Container

5.7 Docker Implementation

5.7.1 Introduction

Having discussed all components, this section will cover how each element of the application is built and deployed using Docker.

5.7.2 Container Structure

A Dockerfile was created for each system component, which to reiterate on Section 4.4.1, are scripts to automate the creation of a Docker image. Listing 13 is an example Dockerfile, containing the definitions used to build the server. As can be seen, Dockerfiles are structured as simple key and value pairs, keys being operations and values being arguments.

```
FROM python:3.9
COPY . /opt
WORKDIR /opt
RUN pip install -r requirements.txt

ENV IN_DOCKER_CONTAINER 1

HEALTHCHECK --interval=30s --timeout=15s CMD python3 health.py

ENTRYPOINT [ "python", "-u", "bonsai.py" ]
```

Listing 13: [Dockerfile] Dockerfile Definitions of the Bonsai Server

The data exporters, being based on Python use the version 3.9 of the official Python image as a base layer. There are seven total steps defined for the exporter Dockerfiles, which, in order, are copying the source code into the image, setting the working directory, installing all requirements, setting a health check command, and setting the entrypoint for the container.

The server, also written in Python has the same steps as the exporter.

The socket, being written in NodeJS, utilizes the version 16 of the official Node image from Docker Hub, a registry for container images (Docker Inc., 2023). Otherwise, the socket is also similar to the exporter and server, as it also copies over the code, installs requirements, and sets an entrypoint.

The frontend features the most complex Dockerfile, as it utilizes a builder container. As has already been discussed in Chapter 2, Dockerfiles can be written to use builder images to compile code, which is then packaged in a container with a more lightweight base. To achieve this, the frontends Dockerfile uses NodeJS as a builder container, which copies over the code, installs dependencies, and builds the VueJS project, resulting in static content. After the build is complete the latest version of the NGINX image is used as the final base image for the container. The compiled website from the build image is copied into the nginx image, along with the NGINX configuration file. Finally, the NGINX executable is set as the entry point, resulting in a container with a total size of just 144 megabytes, one quarter of the size of the original dependencies installed during the build process.

5.7.3 Docker Compose Deployment

The utility of using Docker compose during development comes from the ability to launch elements of the application with just one command. Listing 14 demonstrates this using a simplified version of this project's Docker compose file, only containing the database and server. The database uses the latest RethinkDB image available, while the server specifies that it needs an image to be built from the `bonsai_server` directory. The RethinkDB has a Docker volume mounted under the data directory for data persistence throughout redeployments. The server has the database specified as a dependency, meaning that the server container will only start once the database container has started. Connection between the server and database is specified using the link keyword. Containers also come with the advantage of being "built for failure," this implies that they are designed to handle failures and restart automatically, providing improved reliability. This is specifically defined within the Docker compose file under the restart keyword.

```yaml
services:
  rethink:
    image: rethinkdb:latest
    volumes:
      - rethink_data:/data
    restart: always

  server:
    build: ./bonsai_server
    depends_on:
      - "rethink"
    links:
      - "rethink"
    ports:
      - 50051:50051
      - 50052:50052
    restart: always

volumes:
  rethink_data:
```

Listing 14: [docker-compose.simple.yml] Simplified Version of this Project's Docker Compose File

5.7.4 Health Monitoring

As has been mentioned, Docker containers have built in health checking mechanisms. These commands are defined in the Dockerfile, including an interval and timeout. There are multiple approaches to writing a health check command, as the metric used to decide whether a container is healthy or not, is the shell commands exit signal. Multiple health check approaches were used within this project. The server utilizes a custom Python script health.py, which probes the health servicer gRPC service. The exporters check for specific text in the main process's logs, as there are no points which can be probed. The socket provides a route over its REST API, which returns the health of the service. The frontend is probed over the nginx_status route, a health check mechanism built into NGINX.

5.8 Summary

This projects implementation was successful, in the sense that every major component was created and implemented in a way that achieves the planned goal, while adhering to the standards established during the design process. In addition, a mechanism for the complete deployment of the project was created in the form of Docker images as well as a Docker compose definition file, which automatically deploys the entire system, including networking requirements and volumes for storage.

6 Test Strategy

6.1 Introduction

This chapter will cover the testing methods used to evaluate performance and functionality of the monitoring system, both during development and final deployments. Testing, being an essential part of any project, was done to enable the project to reach all goals, without hitting any major roadblocks along the way.

6.2 Development Lifecycle

6.2.1 Development with Docker

Testing during the development lifecycle has positively affected the development speed of this project. Docker was an essential tool during development, as every component of the system has a dependency to at least one other component. As an example, when working on new exporters, the server and a database can be deployed within seconds by running `docker-compose up -d rethink server`. This command searches the Docker compose file for services named "rethink" and "server," then starts them in the background, as specified by the `-d` flag, standing for daemon. Application logs can then be accessed by running `docker-compose logs -f`. Other than accelerating development, having components of the application isolated as containers comes with several other benefits. With containers, full control over the applications networking is given to the developer, allowing for the simulation of production environments, even during development. Containers will always behave in the same way when started from an image, eliminating any uncertainty. Observability is greatly improved, both through direct access to an application logs and detailed information on a containers resource utilization.

6.2.2 Environment Based Configuration

To further assist the development in different environments, both local and within containers, environment based configuration was implemented for each component. An example of this can be seen in Listing 15, where the address of the database server is set dynamically by checking an environment variable. The environment variable `IN_DOCKER_CONTAINER` is set during the build stage of each Docker container created for this project, thus making it possible for every system component to be dynamically started either locally or within a container, without any changes to the code or configuration being necessary.

```
// NodeJS
var rethinkhost = '127.0.0.1'
if(process.env.IN_DOCKER_CONTAINER == 1){
   var rethinkhost = 'rethink'
}

# Python
if('IN_DOCKER_CONTAINER' in os.environ):
  if(os.environ['IN_DOCKER_CONTAINER']):
    rethink_server = "rethink"
else:
  rethink_server = "localhost"
```

Listing 15: [rethink.js/rethink.py] Example Snippet of Environment Based Configuration

6.3 Continuous Integration and Continuous Delivery

6.3.1 Introduction

The principle behind Continuous Integration and Continuous Delivery, abbreviated as CI/CD, is to help developers deliver high-quality software faster. Continuous integration merges commits from across the repository and automatically builds and tests it. While this is not particularly important in the context of this project, continuous delivery is, as it automatically builds and deploys the software, ensuring a faster pace during development, as deployment issues are able to be found at an early stage.

6.3.2 GitHub Actions

The projects source code is kept in a GitHub repository, both as a backup mechanism, and for version control. GitHub's implementation of CI/CD, which they call "Actions" allow developers to automate parts of the otherwise often manual deployment process of their application. Actions can be configured to be triggered by a multitude of events, such as commits, pull requests, or merges. This project utilizes two workflows, one to build Docker containers of all services, shown in Listing 16, and one to deploy the containers to servers.

```yaml
name: Publish

on:
  push:
    branches: [ "master" ]
  pull_request:
    branches: [ "master" ]

jobs:
  publish:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v3
    - name: publish
      uses: pennsignals/publish_docker-compose@v0.1.0
      with:
        version: 'latest'
        docker_compose: 'docker-compose.build.yml'
        repo_token: "${{ secrets.GITHUB_TOKEN }}"
```

Listing 16: [publish_docker.yml] GitHub Actions Workflow which Builds and Publishes Containers

6.3.3 GitHub Registry

The GitHub Registry is a feature of the GitHub platform, which allows developers to publish and share packages across a multitude of languages. This project utilizes the GitHub Registry to share and store Docker containers created by the aforementioned workflow.

6.3.4 Ansible

Ansible is an open-source automation tool, which aids in the automation of several infrastructure management tasks, such as configuration management, configuration templating, application deployments, and more (Red Hat, Inc., 2023). This project utilized Ansible to deploy the Bonsai monitoring stack and individual exporters to a multitude of hosts automatically. To achieve this, Ansible provides roles, which are units of code, or tasks, used to simplify and automate IT tasks. Roles are a collection of tasks, files, templates, and variables, which can be applied to hosts in a consistent way.

6.4 Testing Infrastructure

6.4.1 Introduction

One of the most important values of this project has always been its real world applicability. This section aims to introduce how the project was tested and deployed on existing infrastructure. Through the Ansible CI/CD pipeline the project was continuously deployed on a collection of servers across Europe.

6.4.2 Hybrid Cloud Infrastructure

The testing of this project greatly benefitted from the access to an existing hybrid cloud infrastructure. The infrastructure consists of two major parts. The first is a private cloud, located in Austria, which includes a firewall, switch, and virtualisation server. The second is Oracle's public cloud, where the computing resources are located in Frankfurt, Germany. The two locations are connected using a site-to-site IPsec VPN, making it possible to securely communicate between the two separate infrastructures, making them a hybrid cloud.

6.4.3 Deployment

As the testing infrastructure is already maintained through Ansible, the only requirement was the creation of Ansible roles for the Bonsai monitoring stack and Bonsai exporters. These roles were then utilized by a CI/CD pipeline to periodically deploy the newest version of each component to each host in the system. Figure 22 is a visual representation of the testing infrastructure, created automatically by the node graph implemented in the Bonsai frontend.

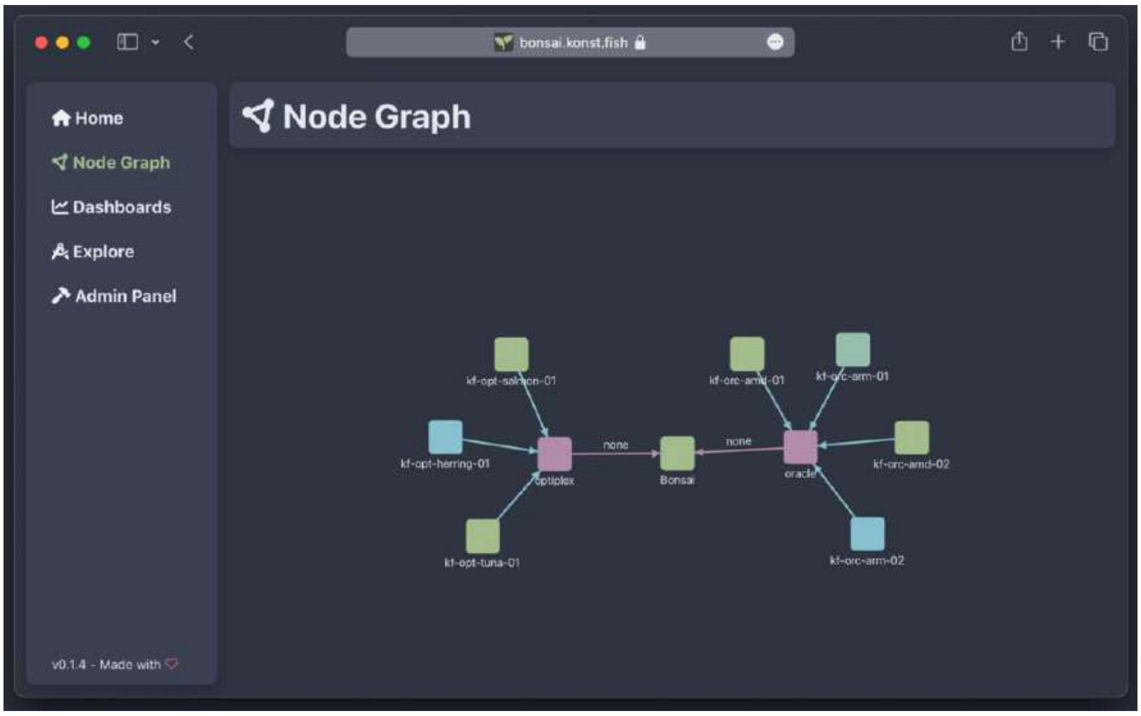


Figure 22: Bonsai Frontend Node Graph View

6.4.4 Conclusions

The successful and continuous deployment to a pre-existing infrastructure greatly impacted this projects development. Several issues were able to be discovered early on, such as asynchrony issues with the server and exporters running on different systems, as well as problems with multiple socket connections from the frontend. Both of these issues would not have been discoverable when just testing the project in a local setting. Having the project deployed in a real setting also provided usability testing, where it proved to be an effective system, even when deployed alongside another monitoring system. As Bonsai is inherently different from other monitoring systems, the case could also be made for parallel deployment, as both deployed monitoring systems were actively used during the testing period, where they complimented each other well.

6.5 Stability Testing

6.5.1 Data Exporter Transfer

To test the data exporters stability and consistency, there are two relevant points to monitor. Scrape time, in other words the time it takes the exporter to gather data from the host machine. Send time, the time it takes the exporter to send data to the server and receive a confirmation reply. The data plotted in Figure 23 was taken from a data exporter running on the “kf-opt-herring-01” virtual machine sending data to the Bonsai stack running on the “kf-orc-arm-02” host, which, as can be seen in Figure 22, communicate over the IPSec VPN.

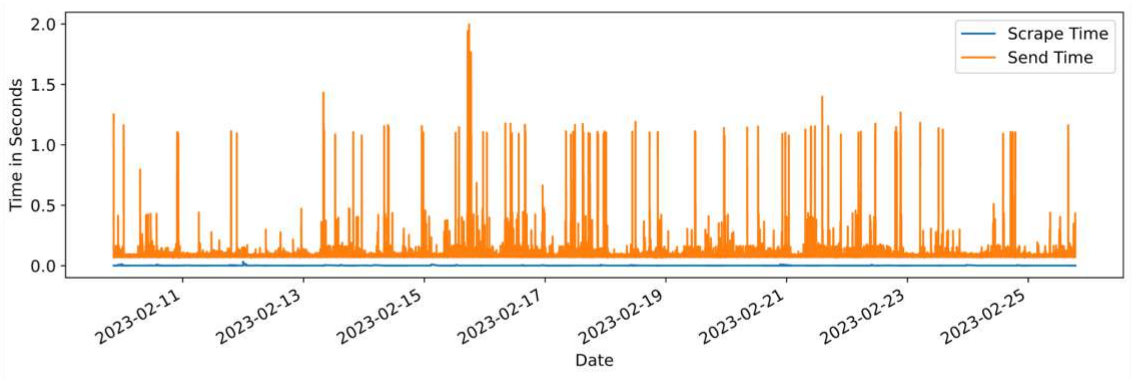


Figure 23: Plot of a Data Exporter’s Scraping and Transfer Time over a Period of 14 Days

While there are no major deviations in scrape time, the transfer time occasionally spikes up to two seconds. In this scenario the exporters metric packages had an average round trip time of 77 milliseconds, which demonstrates the stability of the system. There are many factors that could cause the irregularities in the transfer time, most notably network issues, however considering the timeframe of the shown data, they were few and far between.

6.5.2 Resource Utilization

As mentioned, Bonsai was deployed on an existing infrastructure, where it runs alongside an existing monitoring system, which provided insight into the resource utilization of each component of the system. Figure 24 is a screenshot of the resource usage of the entire Bonsai stack over the course of seven days. As can be seen, the system manages to consistently stay below a maximum of 256 megabytes of memory usage, even when deployed within an actual infrastructure. The system was periodically observed to check for any memory leaks or peaks in usage, none of which occurred during the one month of active deployment and usage.

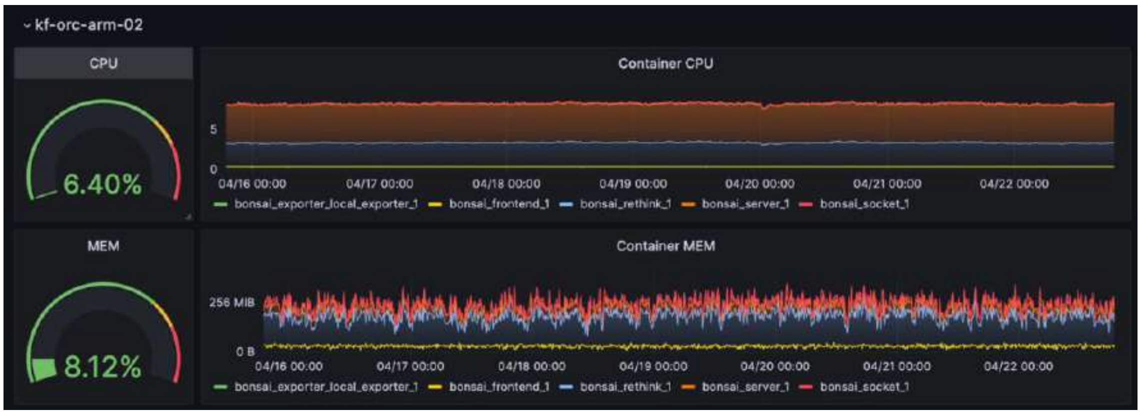


Figure 24: Section of a Grafana Dashboard Observing the Bonsai Stack Deployment

6.6 Functional Testing

6.6.1 Unit Testing

Unit testing describes the isolated testing of individual units, or components, of an application, to ensure that each component is functioning correctly. Due to this projects approach to incremental development, with each component relying on the previous component to be implemented to a fully functional state, unit testing was done in parallel with each component’s implementation. Docker was used to have components completely isolated form the rest of the system to examine their behaviour more closely when connected to the rest of the system. The prototyping of some components, which was done ahead of time of the actual implementation also served as testing scripts in these stages, as certain tasks could be simulated using the already implemented protype components. In part of unit testing, the potential databases were benchmarked, as first discussed in Section 3.2.1, to see which would perform the best within the Bonsai system, as it relies on fast insertions and updates. The results, which can be seen in Figure 25, show RethinkDB to be better suited.

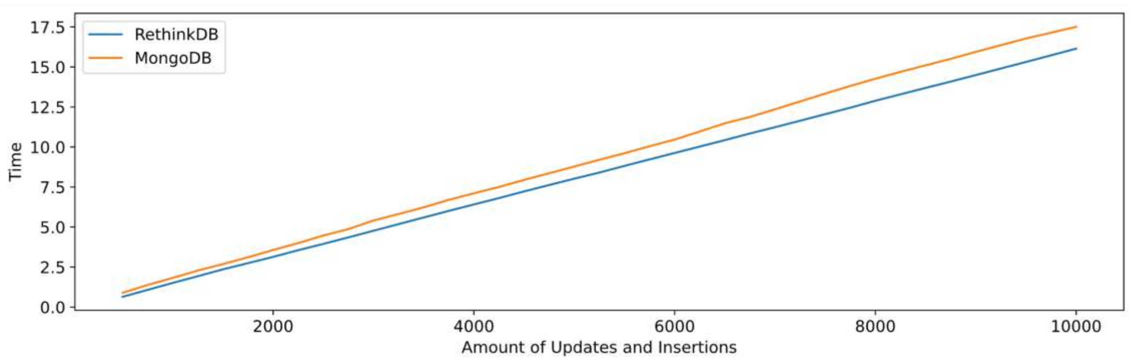


Figure 25: Comparison of RethinkDB and MongoDB using Timed Insertions and Updates

6.6.2 Integration Testing

This project relies heavily on fast and consistent communication between components. Integration testing was completed each time a new component was implemented and appended to the rest of the system. This was possible, even during the development of the individual components, thanks in part due to Docker, which allowed for finished components to be deployed completely isolated from the locally deployed components, creating a realistic testing scenario. The integration testing done in part of this project ensured that the interactions between components are implemented to a high standard.

6.6.3 System Testing

System testing for this project was done as soon as all components were successfully implemented and deployed to the testing infrastructure, which has been discussed in part of Section 6.4. The project was actively used as an alternative to the previously deployed monitoring system to observe the infrastructure, where a few issues affecting finalized deployments were encountered and corrected. To name one of these issues, redeploying the database while the server is actively running would cause the server to accept new metrics from exporters, returning a successful status code, without writing the metrics into the database. This was amended by having exporters re-register with the server once the server detects any out of system changes to the database.

6.6.4 Acceptance Testing

This project underwent acceptance testing after its final deployment on the testing infrastructure. The project was actively used to see if all requirements, originally laid out in Section 3.3, were achieved. In addition, due to the project being deployed in parallel with a more sophisticated monitoring system, actual comparisons to a commercial system could be drawn to judge the projects effectiveness. In addition to the project fulfilling all requirements, it proved to be a satisfactory solution, especially when used in parallel with another monitoring system.

6.7 Non-Functional Testing

6.7.1 Performance Testing

Performance testing was done by actively monitoring the project's resource utilization in regard to processor and memory usage during its deployment on the testing infrastructure, as has been discussed in Section 6.5.2. Comparing the presented results in Figure 24 with the other monitoring system deployed on the testing infrastructure, which consists of Grafana, Prometheus, and a few data exporters, as can be seen in Figure 26, shows that the resource utilization of Bonsai, generally, is lower. Especially when considering memory usage which can go as high as two gigabytes with the Prometheus monitoring system, Bonsai outperforms this system in regard to compute usage, while achieving similar results.

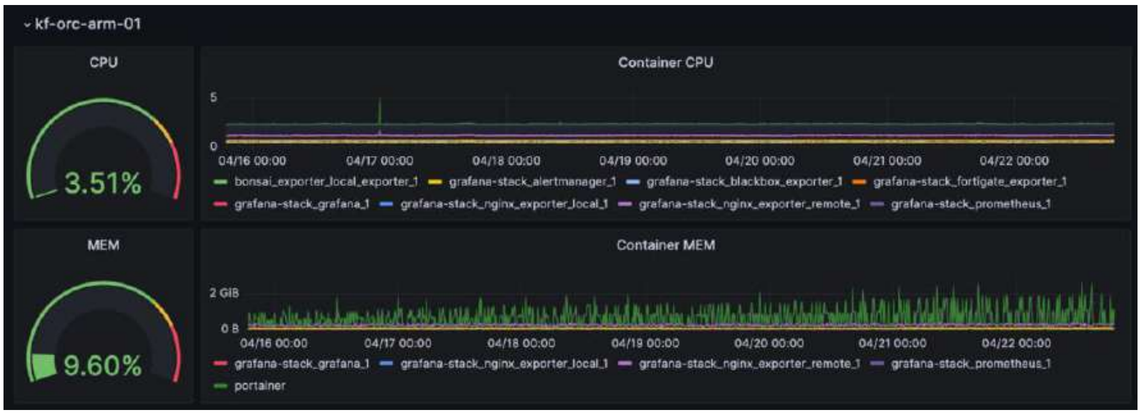

kf-orc-arm-01
CPU
3.51%
Container CPU
bonsai_exporter_local_exporter_1
grafana-stack_alertmanager_1
grafana-stack_blackbox_exporter_1
grafana-stack_fortigate_exporter_1
grafana-stack_grafana_1
grafana-stack_nginx_exporter_local_1
grafana-stack_nginx_exporter_remote_1
grafana-stack_prometheus_1
MEM
9.60%
Container MEM
portainer

Figure 26: Section of a Grafana Dashboard Observing a Grafana Stack Deployment

6.7.2 Scalability Testing

Using Docker compose, the scalability of the project was tested by dynamically deploying a ranging number of exporters to determine the number of exporters the server can handle concurrently. This testing was done on an isolated virtual machine supplied with two gigabytes of memory. In this test case, the machine could handle up to 150 exporters, each providing new metrics every second, without losing a significant amount of performance. This can be seen in Figure 27, which is the activity meter for the metrics table, showing the writes per second in the metrics table.

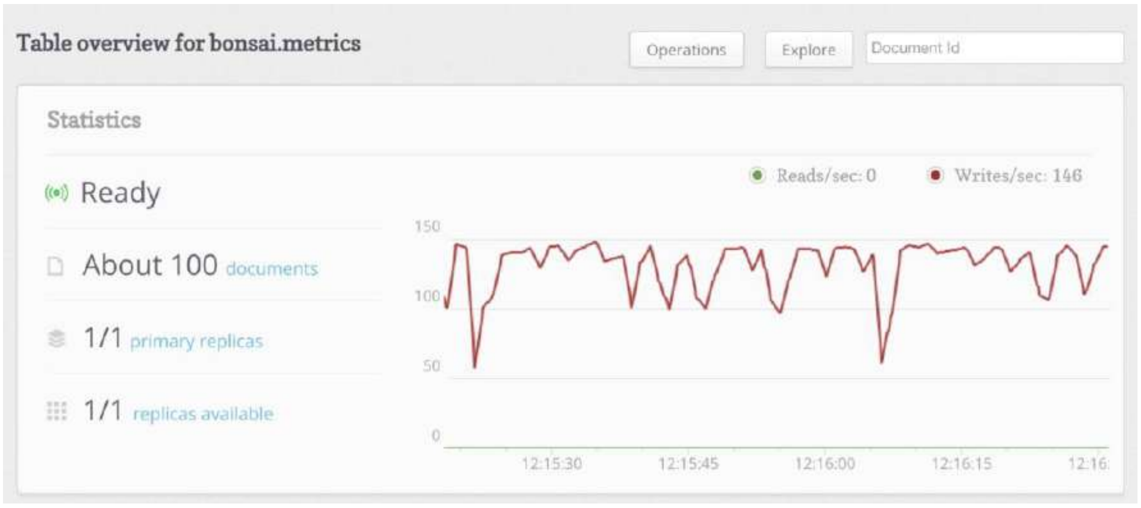


Figure 27: RethinkDB Admin Dashboard Displaying Statistics for the Metrics Table

6.7.3 Usability Testing

Usability testing was done through active usage of the monitoring system on the testing infrastructure. As the component of the system which clients will interact with the most is the frontend, it was the main candidate for the usability testing section. The approach to usability testing was focused on how the system compares to other monitoring systems. Tasks completed in Grafana were compared to how easily they could be achieved on the project's frontend. During testing, no major discrepancies or missing features were discovered, as the system could, within reason, achieve all tasks that could be completed in Grafana.

6.7.4 Security Testing

To ensure the projects security multiple methods were applied. Firstly, some individual components of the project are inherently secure, as they are directly linked to their dependencies, thus having no ports exposed with which they could be accessed. This includes the database, as it is directly linked to the socket and server, as well as the socket, as it is linked to the frontend and exposed through a reverse proxy. The only publicly accessible components of the project are the server, as it needs to be able to receive metrics from exporters, and the frontend, as it needs to be accessible for clients to use. To secure the server, multiple steps were taken such as the implementation of keys for exporters, which act like passwords and can be specified within their individual configurations, as well as the addition of TLS capabilities to both exporters and the server. Security testing was done primarily for the server, by trying to access it without any correct credentials or the corresponding certificate, which was unsuccessful.

6.8 Evaluation

The project went through a number of different tests, both during development and during its final deployment, which ensured, that the monitoring system could be delivered without any major issues or problems. The primary conclusions for an evaluation came from the reliability as well as usability testing, both of which proved that the system could consistently be used in a real deployment scenario. The project therefore fulfils its primary objective, challenging other monitoring systems through its difference to them.

6.9 Summary

This project greatly benefited from the amount of testing it underwent, most importantly, the testing which was completed during development. Issues were discovered and mitigated early on, which positively affected the development cycle and enabled a timely completion of the project. Docker especially was indispensable during development, as it supplied isolated testing even during the earliest stages of implementation. The conclusions made in part of this chapter also were greatly affected by the deployment on actual infrastructure, which was essential to test if the project could stack up with other monitoring solutions, which was found to be true.

7 Evaluation, Conclusions and Future Work

7.1 Project Objectives

The core objectives this project aimed to fulfil were all achieved. A complete monitoring system was created, which is expandable and dynamic, while remaining minimalistic. Each planned component of the system was implemented with all features that were required. A fast and dynamic standard for data transmission between exporters and the ingress server was created. A baseline for exporters was established, as well as a fully functional and modular exporter. An ingress server, capable of authenticating exporters, receiving, and processing metrics, and interfacing with a database to store metrics, all while communicating securely over TLS, was created. A socket that listens for changes in the database to dynamically send them to a client was implemented. Finally, a frontend was created, which includes multiple views to allow clients to interact with the monitoring system. This includes a dashboarding view, on which clients can create custom dashboards. All of these individual components are packaged as Docker images, which allows them to be distributed and deployed on almost any system. When the system is deployed, new exporters can be added dynamically without any changes to the running components. Each component has at least a minor amount of documentation with the code being thoroughly commented, to make future work and expansion of the project not just possible, but also intuitive. In addition, the communication method between exporters and servers, being implemented using gRPC, a transfer standard supporting multiple languages, allows for exporters to be written in almost any language, thus making it possible for any developer to contribute to the project's ecosystem.

7.2 Self-Evaluation

7.2.1 Author Motivation

The initial idea for this project came from a place of discovery. The author, having worked in DevOps and implementing observation tools a number of targets with different monitoring systems noticed the lack of a system like the one created for this project. Initial testing of the feasibility of a project like this was done way ahead of any work completed in part of this project report, to ensure its scope was possible to achieve. This project was used as an opportunity to leverage the knowledge of various programming languages and tools the author accumulated throughout their IT career while gaining insight on new areas. This project, being implemented using a variety of languages and using a multitude of tools to help with both development and deployment, helped deepen that knowledge and lead to a multitude of conclusions made.

7.2.2 Reflection on Prototyping

The plan for this project was created with the initial testing in mind, which helped accurately distribute the implementation time for each planned component and feature required to fully complete the project. Having done this testing way ahead of even laying out the initial requirements has helped shape this projects development, as discoveries on the capabilities of languages and tools were crucial when deciding on the order of work. This primarily refers to what components to implement first and more importantly, in what order certain features should be implemented. While the plan initially laid out using the Gantt chart was not adhered to completely during development, mostly due to the workload caused by other university projects and a few roadblocks encountered during development, the project artefact was still completed in time. At the time of submission, the system is fully functional, and all requirements thought of in the project proposal were completed. The initial set of requirements can be seen in Appendix 1 – Project Proposal.

7.2.3 Reflection on Minimalism

One of the biggest hurdles this project faced was sticking with the minimal theme for every component and transfer method. Primarily due to the time constraint, multiple choices were made that sacrificed the initial vision of minimalism in favour of a working system. The component affected by this the most is the socket, as it provides two major functionalities, the web socket, and REST API, which, when considering the service oriented design approach, should have been two separate components. In addition, the communication method between the web socket and frontend was not planned out completely ahead of its development, which resulted in a lot of unused code and a generally flawed implementation. While it fulfils everything that is required of it, there is room for improvement.

7.2.4 Reflection on the Frontend

The author used this project as a chance to learn frontend development. Starting development for such a major component with minimal prior knowledge definitely posed a challenge and resulted in a dependence on a multitude of libraries to achieve the final implementation. The learning process during development is reflected in the quality of the views background logic, especially when ordered in the order of implementation.

7.2.5 Reflection on the Quality of Code

The project's scope resulted in some sections of the implementation which were either rushed or not implemented to a satisfactory standard. These will all have to be reworked before a public release of the project, to ensure any issues that could arise due to these trade-offs affect any actual deployments of the project.

7.2.6 Lessons Learned

The development of this project resulted in a lot of experience gained and lessons learned. This projects large scope was made possible to achieve in the given timeframe, primarily thanks to the early prototyping that was done to determine which components and features required more time and attention when compared to others. As mentioned in Section 7.2.3, not having a concrete plan for the socket hurt the project's development, as a lot of time was wasted writing code without any blueprint.

7.3 Project Evaluation

This section will cover every component, in order, that was implemented in part of the project to review what was achieved and what changes could be made to improve them.

7.3.1 Server

The server achieves all goals that were laid out during the project planning phase. It is able to receive metrics in a dynamic data format as discussed in Sections 5.2.2 and 5.4.2. The received metrics are then efficiently parsed and filled into the database as shown in Section 5.4.4. The biggest change that should be made would be rewriting the server in a compiled language in favour of Python as, generally, compiled languages will always outperform interpreted languages (Ampomah et al., 2017).

7.3.2 Data Exporters

The primary data exporter was implemented in Python due to the language making it possible to easily expand upon existing code. The original requirement was to create another exporter with similar functionality to the Python exporter, however this was not achieved due to time constraints. Instead, a small proof of concept was created to demonstrate that exporters could be written in any language, export any data and run on any system, which can be seen in Appendix 3 – IOS Gyroscope Exporter. Otherwise, the exporter submitted in part of the project's artefact fulfils all requirements, as it is able to transmit dynamically structured data to the server as shown in Section 5.3.7. In addition, it is configurable, and capable of being expanded by writing new exporter classes, based on an abstract class, as discussed in Sections 5.3.2 and 5.3.4.

7.3.3 Socket

While the socket achieves all tasks it was planned to have, it suffers from code quality, which should be addressed before a final release. The primary changes to be made would be a rework of the web socket communication methods, as currently there is no clear standard for the messages sent between it and the frontend. Apart from this, the socket is fully functional. It is able to provide database listeners to any client, as shown in Section 5.5.4. It also provides the backend of the dashboarding functionality of the frontend over a REST API as demonstrated in Sections 5.5.6 and 5.6.5.

7.3.4 Frontend

The frontend provides all of the functionality that was laid out for it, while having a consistent and easily readable design. There are some sections of its implementation which could benefit from a rewrite, but generally, its performance is not affected by the background logic not being perfect. All views originally planned in Section 4.5.1 were implemented to be fully functional. In addition, a reverse proxy was created in part of the Docker image, which was not part of the requirements, but greatly benefited the project as a whole, as discussed in Section 5.6.8.

7.4 Applicability of Findings to the Commercial World

The final system is straightforward to deploy, especially on infrastructure which is already managed through any infrastructure as code tool. Due to the individual components being packaged in containers, deployment is possible on almost any system. In addition, due to the systems low resource requirements, it can be deployed on the cheapest machines available from public cloud services, without sacrificing performance. The project demonstrated its applicability in a real world deployment, as it was able to run in parallel to a more sophisticated system, and still offer functionality and insights the other system could not. Monitoring is critical for any business planning to succeed in the world of IT, and this project is capable of filling that role.

7.5 Conclusions

Despite the few shortcomings, the system is capable of everything it was planned to achieve. Exporters, which can be written in many languages are able to send any metric to the server. The server is able to authenticate exporters securely and parses incoming information efficiently to save in the database. The socket is able to handle multiple clients at once and provides listeners for metric changes in the database. The frontend has multiple views, all with specific uses and functionality, which all work consistently. Each component is packaged in Docker images and can be distributed and deployed on any host. Other than a few exceptions, the system was kept minimal while still providing powerful functionality. Thanks to the service oriented architecture, components of the system can be removed to transform it into something other than originally planned. This also enables the expansion of the system with other components down the line.

7.6 Future Work

This project was designed with the intention of further development in the future, either from the author, companies, or even a community of contributors to the project.

7.6.1 Exporters

As already mentioned, there is a lot of space for work on exporters. The exporter delivered with this project's artefact already lays the groundwork for different metrics to be collected from servers or even applications simply by creating new Python classes. Due to the communication standard between exporters and servers being open and the server providing a reflection API, further development on exporters in new languages is not only possible but easy to achieve. Due to the system being dynamic in the sense that it can receive any type of metric, future exporters can be written for almost any application.

7.6.2 Dashboarding

Currently, the dashboarding view only supports four types of graphs, which while being enough for the initial implementation, it should be expanded upon to gain more dashboarding capabilities. As mentioned in Section 5.6.4, more chart types can be appended without major effort. Another capability the frontend would benefit from would be the ability to monitor multiple hosts at once, eliminating the need to cycle between different hosts to get the same insight for each.

7.6.3 Components

This project's longevity will primarily be thanks to the sheer potential for additional components to be appended to the system, expanding, and improving it. One of the first components that could make this project truly applicable for business use would be an alerting system of some kind, capable of detecting any metrics which exceed a certain value, which triggers a process to inform a team of the issue. Next to observation, alerting is a crucial part of modern monitoring systems, which is why the project would benefit from a component like this. This component could also be implemented independently of the other components, as the only real requirement would be access to the database.

7.6.4 Comparison to other Systems

The biggest difference Bonsai has to other systems is its commitment to being truly real-time and therefore having no system to store data long term. While this makes it unfit for teams looking for persistent data retention over long periods of time, the system makes up for it with its other capabilities. Bonsai also opted for exporters to push data to the server, as opposed to the server querying data from the exporters, as it is implemented across other monitoring systems. This allows the Bonsai server to run continuously without any changes to its configuration being required to add additional exporters. As was discussed in part of Section 6.4.4, Bonsai in its current state also compliments other systems well, when deployed in parallel. This could be expanded even further upon by creating exporters that take metrics from Bonsai to fill into other monitoring systems, which would enable data retention for the metrics collected by Bonsai, making it more capable without any changes to the existing database structure and methodologies.

To conclude, the project is dynamic, truly real-time, and minimalistic. All of these goals initially laid out in the introduction were achieved through extensive testing and planning. All components were successfully implemented, tested, deployed, and actively used, which enabled the project to be fully completed, adhering to all requirements.

References

Ahmad, M. O., Markkula, J., & Oivo, M. (2013). Kanban in software development: A systematic literature review. *2013 39th Euromicro Conference on Software Engineering and Advanced Applications*, 9–16. https://doi.org/10.1109/SEAA.2013.28

Ampomah, E., Mensah, E., & Gilbert, A. (2017). Qualitative Assessment of Compiled, Interpreted and Hybrid Programming Languages. *Communications on Applied Electronics*, *7*, 8–13. https://doi.org/10.5120/cae2017652685

ApexCharts. (2023). *ApexCharts.js*. Modern & Interactive Open-Source Charts. https://apexcharts.com/

Birje, M., & Bulla, C. (2019, April 22). *Commercial and Open Source Cloud Monitoring Tools: A Review*.

Boettiger, C. (2015). An introduction to Docker for reproducible research. *ACM SIGOPS Operating Systems Review*, *49*(1), 71–79. https://doi.org/10.1145/2723872.2723882

Bolanowski, M., Żak, K., Paszkiewicz, A., Ganzha, M., Paprzycki, M., Sowiński, P., Lacalle, I., & Palau, C. E. (2022). *Eficiency of REST and gRPC realizing communication tasks in microservice-based ecosystems*. https://doi.org/10.3233/FAIA220242

Broadcasting events | Socket.IO. (2022). https://socket.io/docs/v3/broadcasting-events/

Cerny, T., Abdelfattah, A. S., Bushong, V., Al Maruf, A., & Taibi, D. (2022). Microservice Architecture Reconstruction and Visualization Techniques: A Review. *2022 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, 39–48. https://doi.org/10.1109/SOSE55356.2022.00011

Chamas, C. L., Cordeiro, D., & Eler, M. M. (2017). Comparing REST, SOAP, Socket and gRPC in computation offloading of mobile applications: An energy cost analysis. *2017 IEEE 9th Latin-American Conference on Communications (LATINCOM)*, 1–6. https://doi.org/10.1109/LATINCOM.2017.8240185

Chickerur, S., Goudar, A., & Kinnerkar, A. (2015). Comparison of Relational Database with Document-Oriented Database (MongoDB) for Big Data Applications. *2015 8th International Conference on Advanced Software Engineering & Its Applications (ASEA)*, 41–47. https://doi.org/10.1109/ASEA.2015.19

Cyr, D., Head, M., & Larios, H. (2010). Colour appeal in website design within and across cultures: A multi-method evaluation. *International Journal of Human-Computer Studies*, *68*(1), 1–21. https://doi.org/10.1016/j.ijhcs.2009.08.005

dash14.ack. (2023). *V-network-graph*. https://dash14.github.io/v-network-graph/

Docker: Accelerated, Containerized Application Development. (2022, May 10). https://www.docker.com/

Docker Inc. (2023). *Docker Hub Container Image Library*. https://hub.docker.com/

Fatema, K., Emeakaroha, V. C., Healy, P. D., Morrison, J. P., & Lynn, T. (2014). A survey of Cloud monitoring tools: Taxonomy, capabilities and objectives. *Journal of Parallel and Distributed Computing*, *74*(10), 2918–2933. https://doi.org/10.1016/j.jpdc.2014.06.007

Getting Started—Vue.JS. (2022). https://012.vuejs.org/guide/#Introduction

Goldschmidt, T., Jansen, A., Koziolek, H., Doppelhamer, J., & Breivold, H. P. (2014). Scalability and Robustness of Time-Series Databases for Cloud-Native Monitoring of Industrial Processes. *2014 IEEE*

7th International Conference on Cloud Computing, 602–609. https://doi.org/10.1109/CLOUD.2014.86

Grafana Labs. (2022). GitHub. https://github.com/grafana

Grafana: The open observability platform. (2022). Grafana Labs. https://grafana.com/

Greb, S. (2016). *Nord*. https://www.nordtheme.com/

GRPC. (2022). GRPC. https://grpc.io/

Harpreet, K., Jaspreet, K., & Kamaljit, K. (2013). A Review Of Non Relational Databases, Their Types, Advantages And Disadvantages. *International Journal of Engineering Research and Technology (IJERT)*. https://www.academia.edu/44838894/IJERT_A_Review_Of_Non_Relational_Databases_Their_Types_Advantages_And_Disadvantages

Hunter, G. (2019, January 27). *A Comparison Of Serialization Formats*. https://blog.mbedded.ninja/programming/serialization-formats/a-comparison-of-serialization-formats/

Ibrahim, M. H., Sayagh, M., & Hassan, A. E. (2021). A study of how Docker Compose is used to compose multi-component systems. *Empirical Software Engineering*, *26*(6), 128. https://doi.org/10.1007/s10664-021-10025-1

Introduction to gRPC. (2022). GRPC. https://grpc.io/docs/what-is-grpc/introduction/

Introduction to PromQL, the Prometheus query language. (2020). Grafana Labs. https://grafana.com/blog/2020/02/04/introduction-to-promql-the-prometheus-query-language/

JBay Solutions. (2021). *Home—Vue Grid Layout*. A Draggable and Resizable Grid Layout, as a Vue Component. https://jbaysolutions.github.io/

Laskey, K. B., & Laskey, K. (2009). Service oriented architecture. *WIREs Computational Statistics*, *1*(1), 101–105. https://doi.org/10.1002/wics.8

Lopes, G., Bonacchi, N., Frazão, J., Neto, J. P., Atallah, B. V., Soares, S., Moreira, L., Matias, S., Itskov, P. M., Correia, P. A., Medina, R. E., Calcaterra, L., Dreosti, E., Paton, J. J., & Kampff, A. R. (2015). Bonsai: An event-based framework for processing and controlling data streams. *Frontiers in Neuroinformatics*, *9*. https://www.frontiersin.org/articles/10.3389/fninf.2015.00007

Maréchaux, J.-L. (2006). Combining service-oriented architecture and event-driven architecture using an enterprise service bus. *IBM Developer Works*, *12691275*.

Matos, F. F. S. B. de, Rego, P. A. L., & Trinta, F. A. M. (2021). Secure Computational Offloading with gRPC: A Performance Evaluation in a Mobile Cloud Computing Environment. *Proceedings of the 11th ACM Symposium on Design and Analysis of Intelligent Vehicular Networks and Applications*. https://www.academia.edu/67675987/Secure_Computational_Offloading_with_gRPC_A_Performance_Evaluation_in_a_Mobile_Cloud_Computing_Environment

Michel. (2023). *Neumino/rethinkdbdash* [JavaScript]. https://github.com/neumino/rethinkdbdash (Original work published 2014)

Mohammed, L. T., AlHabshy, A. A., & ElDahshan, K. A. (2022). Big Data Visualization: A Survey. *2022 International Congress on Human-Computer Interaction, Optimization and Robotic Applications (HORA)*, 1–12. https://doi.org/10.1109/HORA55278.2022.9799819

Node exporter. (2022). [Go]. Prometheus. https://github.com/prometheus/node_exporter (Original work published 2013)

Node Exporter Full. (2022). Grafana Labs. https://grafana.com/grafana/dashboards/1860-node-exporter-full/

Peinl, R., Holzschuher, F., & Pfitzer, F. (2016). Docker Cluster Management for the Cloud—Survey Results and Own Solution. *Journal of Grid Computing*, *14*(2), 265–282. https://doi.org/10.1007/s10723-016-9366-y

Prometheus. (2022). *Prometheus—Monitoring system & time series database*. https://prometheus.io/

Protocol Buffers. (2022). Google Developers. https://developers.google.com/protocol-buffers

Red Hat, Inc. (2023). *Ansible is Simple IT Automation*. https://www.ansible.com

Reichardt, M., Gundall, M., & Schotten, H. D. (2021). Benchmarking the Operation Times of NoSQL and MySQL Databases for Python Clients. *IECON 2021 – 47th Annual Conference of the IEEE Industrial Electronics Society*, 1–8. https://doi.org/10.1109/IECON48115.2021.9589382

Rianto, R., Rifansyah, M., Gunawan, R., Darmawan, I., & Rahmatulloh, A. (2021). Comparison of JSON and XML Data Formats in Document Stored NoSql Database Replication Processes. *International Journal on Advanced Science Engineering and Information Technology*, *11*, 1150–1156. https://doi.org/10.18517/ijaseit.11.3.11570

Roy-Hubara, N., & Sturm, A. (2020). Design methods for the new database era: A systematic literature review. *Software and Systems Modeling*, *19*(2), 297–312. https://doi.org/10.1007/s10270-019-00739-8

Schlossnagle, T. (2017). Monitoring in a DevOps World: Perfect should never be the enemy of better. *Queue*, *15*(6), 35–45. https://doi.org/10.1145/3178368.3178371

Sevilla Ruiz, D., Morales, S. F., & García Molina, J. (2015). Inferring Versioned Schemas from NoSQL Databases and Its Applications. In P. Johannesson, M. L. Lee, S. W. Liddle, A. L. Opdahl, & Ó. Pastor López (Eds.), *Conceptual Modeling* (pp. 467–480). Springer International Publishing. https://doi.org/10.1007/978-3-319-25264-3_35

Shah, B., Jat, P., & Sasidhar, K. (2022). Performance Study of Time Series Databases. *International Journal of Database Management Systems*, *14*, 1–13. https://doi.org/10.5121/ijdms.2022.14501

Singh, Y. V., Singh, H., & Chauhan, J. K. (2021). Online Collaborative Text Editor Using Socket.IO. *2021 3rd International Conference on Advances in Computing, Communication Control and Networking (ICAC3N)*, 1251–1253. https://doi.org/10.1109/ICAC3N53548.2021.9725782

Śliwa, M., & Pańczyk, B. (2021). Performance comparison of programming interfaces on the example of REST API, GraphQL and gRPC. *Journal of Computer Sciences Institute*, *21*, 356–361. https://doi.org/10.35784/jcsi.2744

Socket.IO. (2023, March 27). *Client API*. https://socket.io/docs/v4/client-api/

Suma, S., & Alqurashi, F. (2019). A comparison study of NoSQL document-oriented database system. *International Journal of Applied Mathematical Research*, *8*(1), 27.

Syromiatnikov, A., & Weyns, D. (2014). A Journey through the Land of Model-View-Design Patterns. *2014 IEEE/IFIP Conference on Software Architecture*, 21–30. https://doi.org/10.1109/WICSA.2014.13

Tedeschi, S., Emmanouilidis, C., Mehnen, J., & Roy, R. (2019). A Design Approach to IoT Endpoint Security for Production Machinery Monitoring. *Sensors*, *19*(10), Article 10. https://doi.org/10.3390/s19102355

The gRPC Authors. (2023a). *Authentication*. GRPC. https://grpc.io/docs/guides/auth/

The gRPC Authors. (2023b). *GRPC Reflection*. https://grpc.github.io/grpc/python/grpc_reflection.html

Thesing, T., Feldmann, C., & Burchardt, M. (2021). Agile versus Waterfall Project Management: Decision Model for Selecting the Appropriate Approach to a Project. *Procedia Computer Science*, *181*, 746–756. https://doi.org/10.1016/j.procs.2021.01.227

Van de Vyvere, B., Colpaert, P., & Verborgh, R. (2020). Comparing a Polling and Push-Based Approach for Live Open Data Interfaces. In M. Bielikova, T. Mikkonen, & C. Pautasso (Eds.), *Web Engineering* (pp. 87–101). Springer International Publishing. https://doi.org/10.1007/978-3-030-50578-3_7

Vlahou, A., Hallinan, D., Apweiler, R., Argiles, A., Beige, J., Benigni, A., Bischoff, R., Black, P. C., Boehm, F., Céraline, J., Chrousos, G. P., Delles, C., Evenepoel, P., Fridolin, I., Glorieux, G., van Gool, A. J., Heidegger, I., Ioannidis, J. P. A., Jankowski, J., … Vanholder, R. (2021). Data Sharing Under the General Data Protection Regulation. *Hypertension*, *77*(4), 1029–1035. https://doi.org/10.1161/HYPERTENSIONAHA.120.16340

Appendix 1 – Project Proposal

Honours Degree Project Proposal

Student Name: David Fischer
Course: BSc (Hons) Computing
Project Title: Bonsai: Lightweight, Fast & Scalable Realtime Monitoring

Project Context

This project's goal is the creation of a full-stack real-time monitoring solution, encompassing exporters for different types of metrics, a backend for metric processing, a database solution for storage & a frontend web application with dashboarding capabilities. To achieve real-time data flow from the exporters to the frontend, this project aims to explore multiple data transfer techniques and database solutions. The ideal outcome would be every aforementioned component, packaged as Docker containers, in a deployment ready state. To ease development Docker will be used as the primary environment, as it provides isolation between containers and cross-platform compatibility as described by (Fairwinds, 2017).

The data exporters should come with a few default metric collection classes, for example, CPU, RAM, disk, and network utilization. They should be designed as expandable as possible and support any programming language. In the long-term, this could allow for an open-source community to form around the project, which would mean quicker development and custom exporter implementations for specific use cases.

To make the collected data presentable, it is planned to implement a frontend website, connected to a socket, which listens for changes in the database and delivers them to the client. The website should feature filtering for tags, hosts, or exporters, as well as dashboarding capabilities, similar to "Grafana". More on "Grafana" and its specific use cases can be found in a blog article by (Knoldus Inc., 2022).

Specific Objectives

- Development of a backend, which can quickly parse information & store it within a database
- Exploration of different database solutions, the most likely option being "RethinkDB" due to the research by (Khedkar & Thube, 2017)
- Creation of data exporters in at least 2 different programming languages
- Establishment of a common communication protocol between exporters and the server using gRPC. The decision to use gRPC instead of other communication methods like REST APIs was made due to its efficiency and speed, researched in the article by (Śliwa & Pańczyk, 2021).
- Creation of a frontend web application with custom dashboarding capabilities and a socket to deliver metric changes to the user

Resources

IDE: VSCode
Testing Deployments: Oracle Cloud, Virtual Machines (Ubuntu/Debian as the Operating System)
Languages: Python, NodeJS, Go
Tools: Docker, Kubernetes, gRPC, socket.io, RethinkDB

Potential Ethical or Legal Issues

Most legal issues that this project would encounter stem from potential data breaches. Generally, depending on the types of metrics being collected, the General Data Protection Regulation should not apply. To ensure data security is upheld, measures like TLS encryption between the exporters and the backend will need to be put into place. Additionally, the frontend should be behind authentication and HTTPS encryption. The database would also need security measures, the most obvious being not opening the database port or using nonstandard account names, as well as other more complex measures like described by (Bertino & Sandhu, 2005).

Potential Commercial Considerations - Estimated costs and benefits

Since this project is primarily a software solution, not much hardware is needed to test it. Thanks to Docker, the entire project should be able to be spun up within minutes on any laptop or computer. To test the feasibility of the project Oracle Cloud's "Always Free" resources offer multiple resources which facilitate a good testing environment. This includes 4 OCPUs and 24GB of RAM which can be split up and assigned to up to 4 Virtual Machines across different Virtual Networks & Availability Zones. An extensive list of the resources the (Oracle Corporation, 2022) offers in their "Always Free" tier can be found on their website.
Since this project should work for a firm's entire network, I plan to connect the Oracle Cloud Instances to my home server using an IPsec Tunnel, forming a kind of Hybrid Cloud and allowing for testing in a real-world situation.

Proposed Approach

The following diagram shows the proposed application structure, the flow of data being from top to bottom. The exporters, shown running across multiple operating systems and languages, feed data into the server, which parses it and writes it into the database. From there, the data is picked up by the socket and delivered to any clients currently observing the frontend website.

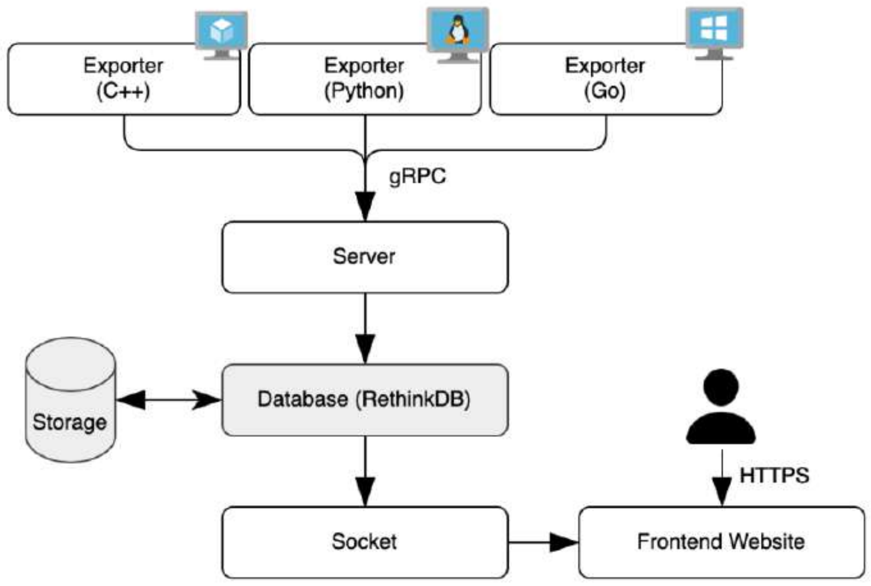


Figure 1: Proposed application structure

Following up on the diagram, this project could be implemented using a top to bottom approach, top being the server and exporters, bottom the frontend.

1. Backend server & database structure
    a. Creation of a gRPC protocol & data structures
    b. Familiarization with "RethinkDB"
    c. Creation of database structure
2. Data exporters (in at least 2 languages) with a modular structure
    a. Creation of gRPC client code
    b. Creation of a modular "pluggable" structure for additional monitoring endpoints
3. Basic socket.io implementation
    a. Implementation of socket.io client handling
    b. Creation of a database listener, which pushes new changes over a socket.io tunnel
4. Frontend implementation
    a. Implementation of client-side socket.io
    b. Implementation of custom dashboarding
5. gRPC Proxy & TLS passthrough implementation for monitoring across Multiple Networks

References

Bertino, E., & Sandhu, R. (2005). Database security - concepts, approaches, and challenges. *IEEE Transactions on Dependable and Secure Computing*, 2-19.

Fairwinds. (2017, September 28). *The Benefits of using Docker for Development and Operations*. Retrieved from Medium: https://medium.com/uptime-99/the-benefits-of-using-docker-for-development-and-operations-2c5256ad89bc

Khedkar, S., & Thube, S. (2017). Real Time Databases for Applications. *IRJET Journal*, 2079-2082.

Knoldus Inc. (2022, May 17). *Getting started with Grafana and Prometheus*. Retrieved from Medium: https://medium.com/@knoldus/getting-started-with-grafana-and-prometheus-4176c1408396

Oracle Corporation. (2022). *OCI Cloud Free Tier*. Retrieved from Oracle OCI: https://www.oracle.com/uk/cloud/free/

Śliwa, M., & Pańczyk, B. (2021). Performance comparison of programming interfaces on the example of REST API, GraphQL and gRPC. *Journal of Computer Sciences Institute*, 356-361.

Appendix 2 – Technical Plan

Honours Degree Technical Plan

Name: David Fischer
Course: BSc (Hons) Computing
Supervisory: Matthew Bates

Title

Bonsai: Lightweight, Fast & Scalable Realtime Monitoring

Summary

This project aims to create a fully featured monitoring stack, capable of collecting data of almost any origin, type, or structure, storing said data within a database and displaying it on a real-time dashboarding solution in the form of a web application.

The project will include multiple modules to achieve its goal:

- Data Exporters
  - Collect metrics from diverse data sources, such as host data and applications.
  - Transfer collected metrics in a predefined format using a secure & fast protocol
- Backend Server
  - Listen for incoming metrics from data exporters
  - Ensure the exporters are authorized to send metrics to the server
  - Store incoming metrics in a database
- Socket Application
  - Listen for changes in the Database
  - Transfer changes to the web application using a web socket
- Frontend Web Application
  - Display metrics collected by the data exporters
  - Make metrics presentable using custom dashboarding and filtering capabilities

See the diagram below to get an idea of how the project will be structured and how metrics will flow through the system.

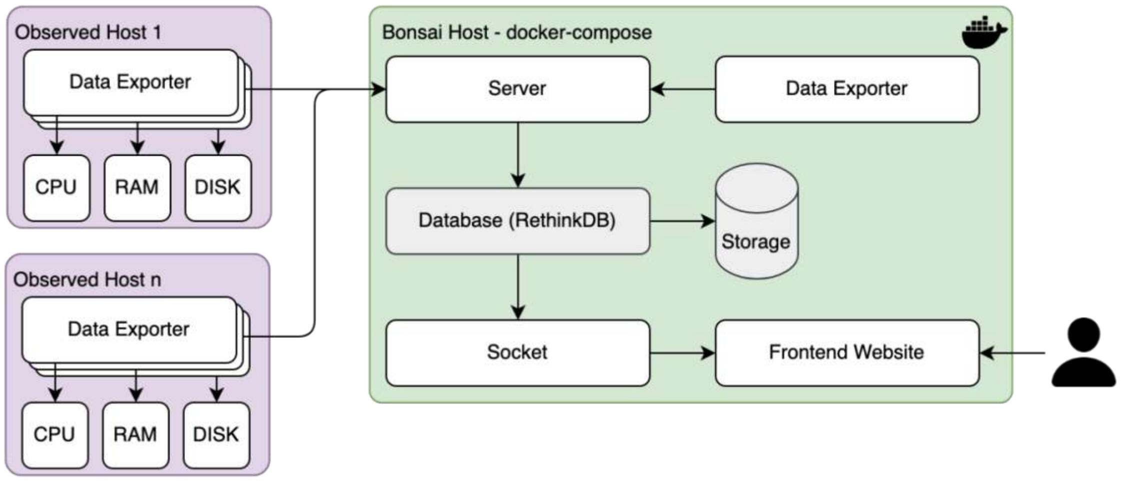


Figure 1: Diagram of proposed project structure

1

Deliverables

The deliverables for this project will be each of the aforementioned modules with their key functionalities implemented as pre-packaged Docker Images for easy deployments, explained by (Singh & Singh, 2016). These images will be deployed using a docker-compose stack to allow for easier testing and local deployments as described by (Ibrahim, Sayagh, & Hassan, 2021). As well as the Docker Images, the code of each module is included.

In addition, a publicly accessible version of the project will also be provided, hosted using (Oracle Corporation, 2022) Cloud Compute Instances. This public version should have at least five Data Exporters feeding metrics into the system, to demonstrate all features implemented in this project.

Since it is planned to publish this project as an open-source monitoring solution after its completion, the deliverables will also include documentation on the gRPC structure and all of the components. Since the data exporters are planned to be configurable, documentation on the configuration will also be provided. In addition, due to the project using git as version control, multiple releases to demonstrate the projects growth over time will also be included.

Constraints

The project must adhere to the following deadlines, defined by the CO3808 Project (Double) course.

Deadline	***Form of Assessment***	***% Weighting***	***Size of Assessment/Duration***
31st October 2022	Technical Plan	5%	Around 4 pages (use template)
5th December 2022	Background and Related Work	10%	Around 10 pages (use template)
30th January 2023	Show and Tell	Formative	Demo/presentation to supervisor
13th March 2023	Project Artefact	40%	Practical work of some sort using computing technology
27th March 2023	Draft Project Report	Formative	Around 50 pages
24th April 2023	Project Report	35% (+10%)	Around 50 pages
2nd May 2023	Project Video	Formative	180-240 seconds
2nd May 2023	Project Poster	10%	A1 Poster

The project faces no financial constraints, as it is a software solution. The hosting costs for test implementations will be covered by Oracle Clouds "Always Free" tier. The database used for storing metrics, (RethinkDB, 2022), is open-source software.

Key Problems

The main difficulties that this project will face are related to the code that will have to be implemented in order to get all modules to a working point.

This project serves as an opportunity to explore different programming languages, which primarily affects the data exporters. The idea would be to make sure that they are expandable and implementable in almost any language. To achieve this, gRPC (Braun, Cheng, Dowey, & Wollert, 2020) will be used to enable fast and secure communication between the exporters and the server.

To keep traffic between the web socket and frontend minimal, the socket will have to be programmed in a way to allow the frontend to decide which data is transmitted from the database. The RethinkDB API features a command to listen to changes in a table (RethinkDB, 2022). This command should also have the possibility to filter for specific data, which will be the key to implementing the dynamic data transmission. For example, this would allow the frontend to only request metrics from a specific host, several hosts with specific tags, or hosts exporting a specific type of metrics.

To display the metrics being scraped in a presentable format, the frontend will implement dynamic dashboarding features. To create this feature, a charting library as well as a method to display the charts in grids will have to be implemented. As I am unfamiliar with frontend development using JavaScript Frameworks like VueJS or React, this feature is part of the key problems.

System and Work Outline

The project has the following key objectives, which will also be worked on in the following order:

- Implementing a common gRPC communication protocol between data exporters and the server
- Writing an exporter to scrape data from the host machine (Metrics like CPU/RAM usage will suffice for the start)
- Implementing a database structure using "RethinkDB"
- Writing a server to receive metrics and interface with "RethinkDB"
- Writing a web socket to interface with "RethinkDB", listen for changes & send them to a website
- Writing a website capable of receiving data from the web socket and displaying it

This development cycle will be supported by Docker to allow for isolated testing of components and quicker delivery due to docker-compose stacks being modular and able to deploy only certain containers. To aid the development cycle, CI/CD will be implemented. (GitHub, 2022) features a CI/CD implementation, which they call "Actions". These can be configured to run on every commit to build the project, deploy it, and run tests. Having a deployment for every commit doesn't only facilitate easy testing, it also allows for mistakes to be caught in earlier stages of development, as well as other methods described by (Shahin, Ali Babar, & Zhu, 2017).

Since the project aims to be a fully features monitoring environment, a proper testing environment would need multiple servers and exporters pushing data into the system. To achieve this, Ansible by (Red Hat Inc., 2022) will be used to provision multiple Compute Instances and deploy containers housing data exporters.

Going into more details on the key objectives mentioned above, the primary languages used will be Python and NodeJS. The data exporters and server will be written in Python, due to its easy syntax and wide range of community modules. The web socket and frontend will be written in NodeJS, due to its proficiency when developing frontend-facing software.

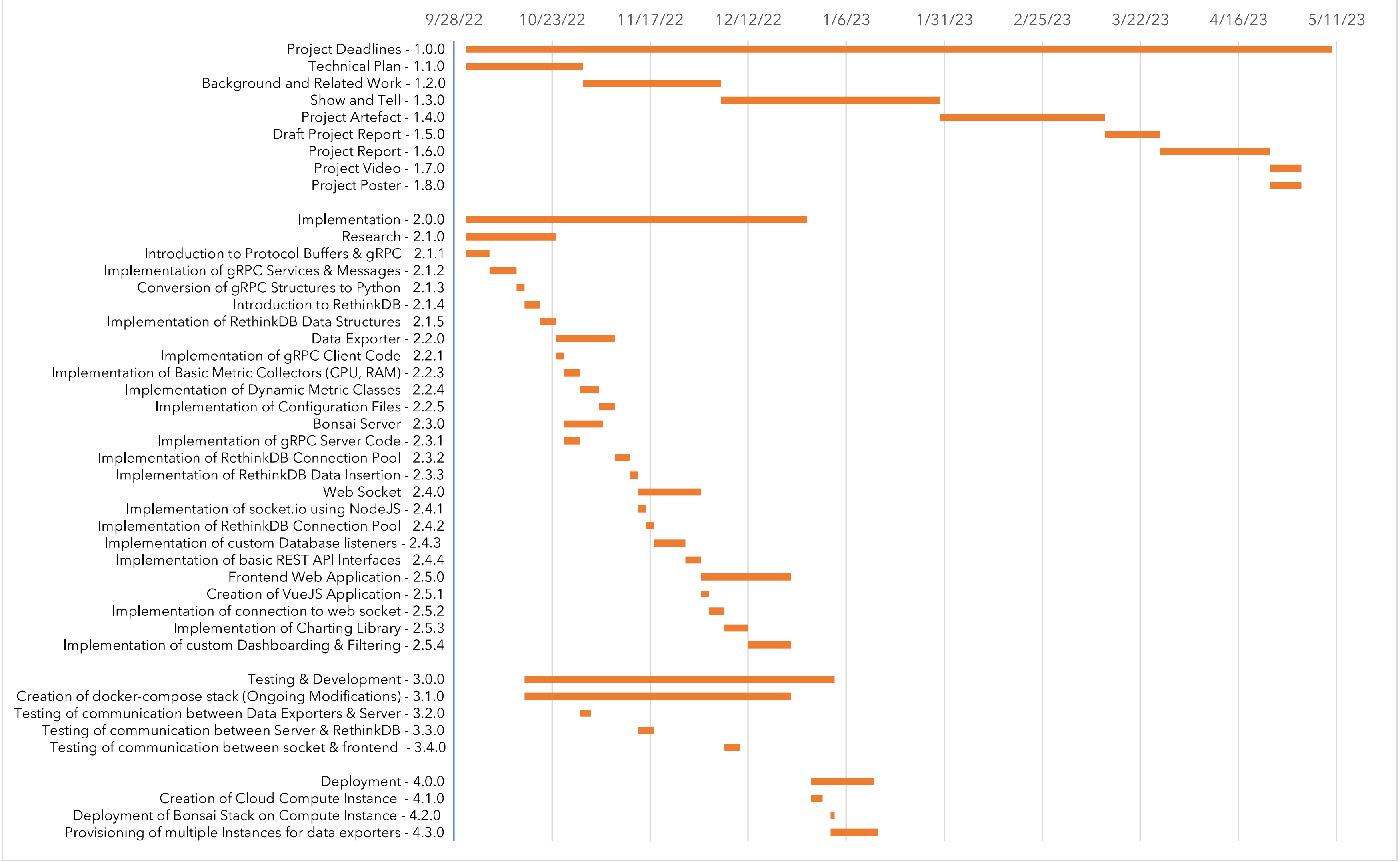
To implement fast and secure communication between data exporters and the servers, gRPC will be used. The open-source high performance (Śliwa & Pańczyk, 2021) Remote Procedure Call framework features built-in TLS client certificate verification, which makes ensuring data exporters are allowed to push data to the server possible without any other form of authentication.

RethinkDB will be used as the real-time database solution due to its extensive API available in both Python and NodeJS. Besides the well documented API, it outperforms other similar dynamic databases like MongoDB for real-time tasks, researched by (Khedkar & Thube, 2017).

Project Activities

See the Gantt chart below, as well as a description of its most critical tasks.

Most noticeable on the Gantt chart is the gap between the Deployment stage and the Project Artefact submission deadline. This is to allow for exploration of different data exporters, including different languages, types, and data.


9/28/22
10/23/22
11/17/22
12/12/22
1/6/23
1/31/23
2/25/23
3/22/23
4/16/23
5/11/23
Project Deadlines - 1.0.0
Technical Plan - 1.1.0
Background and Related Work - 1.2.0
Show and Tell - 1.3.0
Project Artefact - 1.4.0
Draft Project Report - 1.5.0
Project Report - 1.6.0
Project Video - 1.7.0
Project Poster - 1.8.0
Implementation - 2.0.0
Research - 2.1.0
Introduction to Protocol Buffers & gRPC - 2.1.1
Implementation of gRPC Services & Messages - 2.1.2
Conversion of gRPC Structures to Python - 2.1.3
Introduction to RethinkDB - 2.1.4
Implementation of RethinkDB Data Structures - 2.1.5
Data Exporter - 2.2.0
Implementation of gRPC Client Code - 2.2.1
Implementation of Basic Metric Collectors (CPU, RAM) - 2.2.3
Implementation of Dynamic Metric Classes - 2.2.4
Implementation of Configuration Files - 2.2.5
Bonsai Server - 2.3.0
Implementation of gRPC Server Code - 2.3.1
Implementation of RethinkDB Connection Pool - 2.3.2
Implementation of RethinkDB Data Insertion - 2.3.3
Web Socket - 2.4.0
Implementation of socket.io using NodeJS - 2.4.1
Implementation of RethinkDB Connection Pool - 2.4.2
Implementation of custom Database listeners - 2.4.3
Implementation of basic REST API Interfaces - 2.4.4
Frontend Web Application - 2.5.0
Creation of VueJS Application - 2.5.1
Implementation of connection to web socket - 2.5.2
Implementation of Charting Library - 2.5.3
Implementation of custom Dashboarding & Filtering - 2.5.4
Testing & Development - 3.0.0
Creation of docker-compose stack (Ongoing Modifications) - 3.1.0
Testing of communication between Data Exporters & Server - 3.2.0
Testing of communication between Server & RethinkDB - 3.3.0
Testing of communication between socket & frontend - 3.4.0
Deployment - 4.0.0
Creation of Cloud Compute Instance - 4.1.0
Deployment of Bonsai Stack on Compute Instance - 4.2.0
Provisioning of multiple Instances for data exporters - 4.3.0

One of the most critical tasks in this Gantt chart outlines would be 2.1.0 Research, as it lays out the foundation this project will be built upon. The Research task serves as a testing stage to write testing code and functions to serve as a small proof of concept.

The other subtasks in 2.0.0 Implementation describe the rest of the implementation, following a top-to-bottom approach starting with the Data Exporters and ending with the Frontend.

Risk Analysis

The table below outlines the risks this project might face on a scale from 5 (High) to 1 (Low).

Risk	***Severity***	***Likelihood***	***Action***
Data loss due to unforeseen circumstances	5	1	Implement version control using git on platforms like GitHub or GitLab
Inability to meet predefined project deadlines	3	2	Ensure project assessments are done a few days before the predefined deadline
Inability to implement project features as planned	5	2	Keep alternatives to solutions in mind, as described in the Options chapter
Risk of bad actors inserting data into the public instance hosted as part of the artifact	3	3	Ensure gRPC traffic is encrypted & secured using TLS Certificates
System not scaling appropriately with more data exporters being added	4	3	Ensure the code is asynchronous and readable for easier changes and expandability

Options

This project would benefit most from a Waterfall approach, similar to the one already outlined in the Gantt chart's structure. As the project will be completed by one person, there is no further need for any more advanced project management implementations. In addition to the Gantt chart, a Kanban board (Rehkopf, 2022) will be used to create a more visually pleasing presentation of the project's tasks. This also provides the ability to split the primary tasks from the Gantt chart into more granular tasks.

Should RethinkDB turn out not to be a viable real-time database solution in the context of this project, there are other open-source databases available for use. These would include the first alternative of MongoDB, which has similarities to RethinkDB in its API and dynamic structure.

gRPC might turn out to be too complex to implement in this project's dynamic vision with its restrictive number of datatypes available for transfer. The best alternative to gRPC would be a REST API, which would make transferring dynamic data in a dictionary format possible.

As already mentioned, gRPC might be too much effort to implement in multiple programming languages. To circumvent this, gRPC could only be implemented in one language to then serve as a middle layer between the server and a REST API. This would be a separate module for the project, which allows for easier implementation of different exporters using just requests to push data to the server.

Potential Ethical or Legal Issues

At its core, the project does not have any legal constraints as the data being collected stems from machines and not users. This means that the GDPR does not apply to the system should it be deployed in this manner.

The modular approach with exporters might lead to the development of user monitoring, for example, scraping data from fitness devices, or an exporter to monitor which user is currently connected to a Remote Desktop server. Depending on the data being scraped this could lead to potential legal or ethical issues, should the project be used in this fashion, which is not intended.

Commercial Analysis

The table below outlines the different costs and benefits affecting this project. Due to the project being developed using mostly open-source or free tools, the only cost is the labour.

Factor name	***Description***	***Cost/ Benefit***	***Estimated Amount***	***Estimate of when paid***
Labour	Cost of the development cycle per hour.	Cost	500 Hours x 25£ = 12.000£	As soon as all deliverables are fulfilled
RethinkDB	Open-Source Database	Cost	0£	N/A
Oracle Cloud	IaaS Provider	Cost	0£	N/A
GitHub Repository	Backup code storage	Cost	0£	N/A
Docker & additional Software	Testing & Deployment	Cost	0£	N/A

Employability Contribution

For my final year project at the University of Central Lancashire, I developed a full-stack monitoring application. The inspiration for the project was my previous experience working with monitoring solutions like Grafana and Prometheus. These systems, while great for long term metric storage, usually come with the downside of data pollution and bloated exporters. I wanted to implement a real-time solution solely focused on delivering metrics from a host to a dashboard as fast as possible. The challenges I faced during development were making sure data exporters were as slim and fast as possible while still maintaining expandability. This also affected the database design, as I wanted the exporters to be able to scrape any metrics desired by the user.

The skills and toolset I acquired during development primarily revolve around working on a full-stack system. Writing frontend and backend code and deploying the software packaged as containers on Cloud Provider Infrastructure. To make this process of provisioning the software more streamlined CI/CD pipelines were implemented. Due to the project being made up of multiple modules, Docker was used to isolate each component and docker-compose was used to provision the entire software stack. This isolated approach also made the development cycle more efficient, as it allows for essential components like the server and database to be ran as containers, while working on components like the data exporters.

Skills acquired or used during development:

- Languages:
  - Python
  - NodeJS
  - Additional: Go, C++, Swift
- Toolset
  - Docker + docker-compose
  - git + CI/CD
  - gRPC
  - RethinkDB
- Cloud Providers:
  - Oracle Cloud
  - Self-Hosted
- Provisioning
  - Ansible

References

Braun, S., Cheng, C.-T., Dowey, S., & Wollert, J. (2020). Survey on Security Concepts to Adapt Flexible Manufacturing and Operations Management based upon Multi-Agent Systems. *2020 IEEE 29th International Symposium on Industrial Electronics*, 480-484.

GitHub. (2022). *Automate your workflow*. Retrieved from GitHub Features: https://github.com/features/actions

Ibrahim, H., Sayagh, M., & Hassan, A. (2021). A study of how Docker Compose is used to compose multi-component systems. *Empirical Software Engineering volume 26*, 128.

Khedkar, S., & Thube, S. (2017). Real Time Databases for Applications. *International Research Journal of Engineering and Technology (IRJET)*, 2078-2082.

Oracle Corporation. (2022). *Oracle OCI*. Retrieved from OCI Cloud Free Tier: https://www.oracle.com/uk/cloud/free/

Red Hat Inc. (2022). *Ansible*. Retrieved from Automation for everyone: https://www.ansible.com/

Rehkopf, M. (2022). *Atlassian*. Retrieved from What is a kanban board? : https://www.atlassian.com/agile/kanban/boards

RethinkDB. (2022). *Home Page*. Retrieved from RethinkDB: https://rethinkdb.com/

RethinkDB. (2022). *JavaScript ReQL command reference*. Retrieved from ReQL command: changes: https://rethinkdb.com/api/javascript/changes/

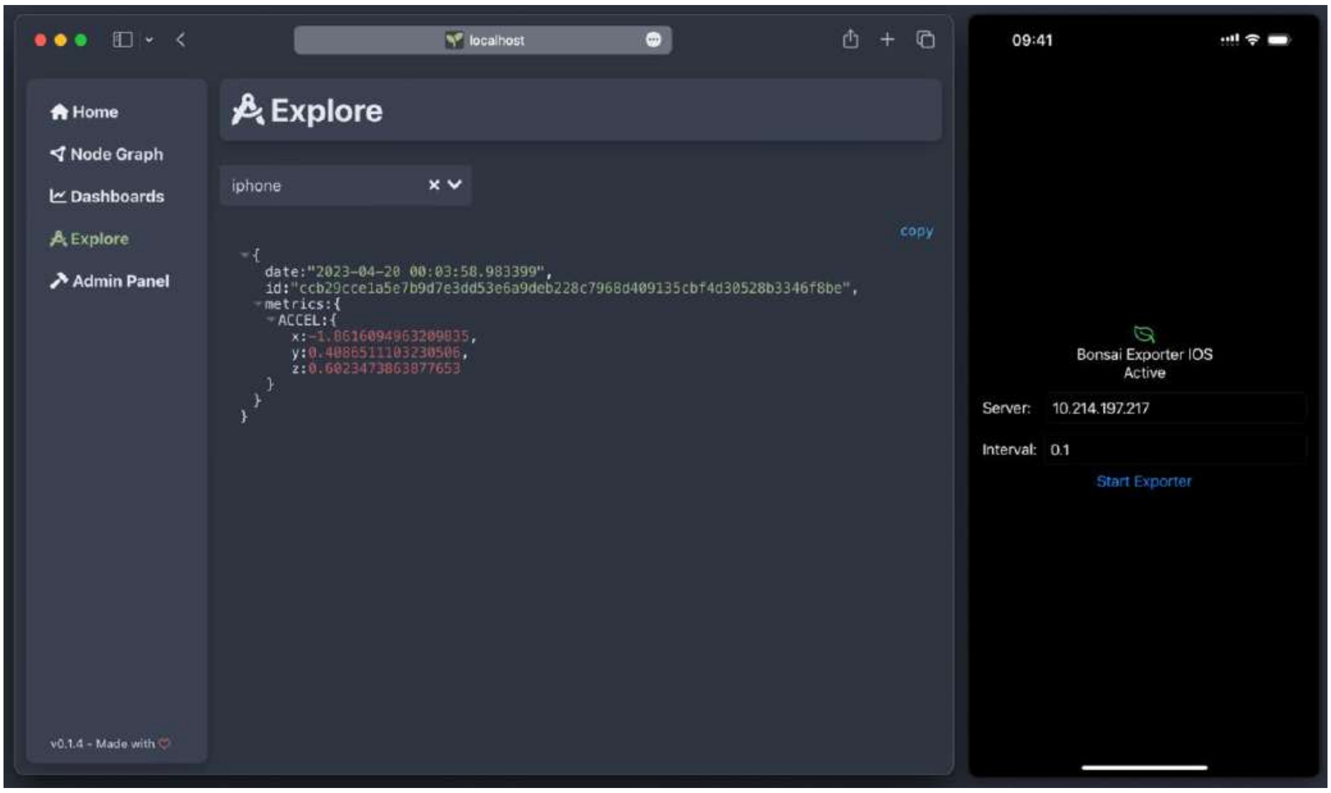
Shahin, M., Ali Babar, M., & Zhu, L. (2017). Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices. *IEEE Access Vol. 5*, 3909-3943.

Singh, S., & Singh, N. (2016). Containers & Docker: Emerging roles & future of Cloud technology. *2016 2nd International Conference on Applied and Theoretical Computing and Communication Technology*, 804-807.

Śliwa, M., & Pańczyk, B. (2021). Performance comparison of programming interfaces on the example of REST API, GraphQL and gRPC. *Journal of Computer Sciences Institute*, 356-361.

Appendix 3 – IOS Gyroscope Exporter

The IOS gyroscope exporter was written as a proof of concept, to demonstrate the projects capability to export any type of metric, in any language, from any type of device. In this case, the pitch, yaw, and row data from an iPhone's internal gyroscope are continuously queried and relayed to the monitoring system, which is capable of receiving this data within increments as small as 0.05 seconds. The figure below shows the explore page of the Bonsai frontend receiving the gyroscope information from the IOS exporter app, which is shown on the right.



```
{
  date:"2023-04-28 00:03:58.983399",
  id:"ccb29cce1a5e7b9d7e3dd53e6a9deb228c7968d409135cbf4d30528b3346f8be",
  metrics:{
    ACCEL:{
      x:-1.8616094963209835,
      y:0.4086511103230506,
      z:0.6023473863877653
    }
  }
}
```

Server: 10.214.197.217

Interval: 0.1