Monitoring Distributed Systems with Open-Source Software

A Major Qualifying Project Report
In partial fulfillment of the requirements for the Degree of Bachelor of Science

Submitted by: Johnathan A’Vant
                Khuyen Cao
                Chad Underhill

Submitted to: Professor Mark Claypool
                Worcester Polytechnic Institute

Sponsoring Organization: A large networking company

Sponsors: Steve Tufts
                Lily Dayton

Date: March 2nd, 2019

This report represents the work of WPI undergraduate students submitted to the faculty as evidence of completion of a degree requirement. WPI routinely publishes these reports on its website without editorial or peer review. For more information about the projects program at WPI, please see http://www.wpi.edu/academics/ugradstudies/project-learning.html
Abstract

Developing on distributed systems increases the maintenance requirements of software projects. Monitoring software can help systems administrators decrease maintenance costs, but existing enterprise solutions are expensive. The goal of this project was to design and build a monitoring dashboard for Company’s distributed development platform using open-source services. We designed and implemented a prototype monitoring dashboard to visualize the health of a typical distributed system. We then tested the prototype dashboard with Company engineers and determined that our approach to implementing an open-source monitoring system could not fulfill the enterprise requirements envisioned by Company.
# Table of Contents

**Introduction**  
4

**Background**  
8  
Distributed Orchestration and Distributed Platform Manager  
8  
Metrics and Instrumentation  
9  
Monitoring  
10  
  Overview  
11  
  The History of Monitoring  
12  
  Cloud Native Monitoring  
13  
  Time Series Databases  
14  
  Whitebox vs. Blackbox Monitoring  
14  
Technology Stack  
15  
  Amazon Web Services  
15  
  Prometheus  
16  
  Node Exporter  
17  
  Grafana  
17  
  ZooKeeper/Kafka  
19  
  JMX Exporter  
19  
  Prometheus Alert Manager  
19  
  etcd  
20

**Related Work**  
21  
  Splunk  
21  
  Datadog  
23

**Methodology**  
25  
  Gathering Project Requirements  
25  
  Determining Key Metrics in System Health  
28  
  Designing and Configuring a Prototype Monitoring Dashboard  
29  
    Determining an Aggregation Process  
30  
    Choosing Alerts and Thresholds  
32  
    Designing the Prototype Monitoring Dashboard  
33  
  Testing the Prototype Monitoring Dashboard  
35

**Implementation**  
37  
  Amazon Web Services (AWS) Instance Provisioning  
38  
  Prometheus, Grafana, and Node Exporter Setup and Configuration  
39
1. Introduction

A distributed system is a group of computers working in unison to appear as a single computer to the end user. The machines have a shared state, operate concurrently, and can fail independently without significantly affecting the entire system. These technologies make it easier for software development teams to build complex, highly-scalable architectures on any cloud-based computing platform.

Company is a multinational computer networking corporation headquartered in Sunnyvale, CA. Company researches and develops enterprise-grade networking hardware devices and software, using distributed computing and orchestration technologies to improve their software development and deployment processes. Developing on distributed systems increases the technical complexity of software projects, and thus increases maintenance requirements; development operations (devops) engineers have to keep track of a large set of components and services within each software package to ensure that their systems are running properly.

Traditional devops monitoring processes involved individual “check-in” interactions with each component within a distributed software system, typically done via command-line. This approach required devops engineers to frequently check-in with software components, as few systems were in place to notify them if something went wrong. The increase in software complexity—aided by the rising prominence of containerization such as Linux, Docker and distributed orchestration technologies such as Kubernetes (Churchman, 2018)—has rendered traditional devops techniques unsustainable; it now takes longer to identify where problems
within software systems occur, since there are many distributed services interacting with and relying on each other.

Monitoring software makes it easier for systems administrators and devops engineers to see how system components are functioning, especially at a large scale. Instead of requiring specialists to individually check hundreds to thousands of services in a cloud-based computing cluster, monitoring systems automatically perform these checks and pull this information into a centralized location. Once collected, a visualization dashboard displaying the synthesized information can provide insight into computer system performance and activity, including operating system resource use, user activity, system processes, and other related components.

Unfortunately, there are not many enterprise-grade monitoring solutions that are available to license—and those that perform well have licensing fees that exceed the budget for Company and some of their customers. Top enterprise platforms include Datadog and Splunk (CrunchBase, 2019). The Datadog Enterprise platform specializes in providing lightweight monitoring with minimal a minimal software footprint, and costs a minimum of $2,300 per month ($23/host for a minimum of 100 hosts). Splunk Enterprise works best with large amounts of monitoring data spread across many applications, and costs $150 per GB of data processed, per month (with a minimum of $1800 per year).

Many of the services in Company’s technology stack produce metrics that detail internal processes. These metrics contain information like processing load, network activity, system errors, service availability, and more. Open-source aggregation and visualization platforms like Prometheus and Grafana have the potential to monitor these services and provide analysis of their data for little to no cost. Devops engineers and system administrators could then use these
dashboards to gain a better understanding of the system’s health and determine where systems operations are going—or are likely to go—wrong. This has the potential to save Company engineers and administrators time during project deployment and maintenance.

Our goals for this project include researching modern monitoring concepts to better understand what makes a useful dashboard, building a dashboard to monitor part of Company’s technology stack, and testing our monitoring dashboard with Company engineers to determine how it could be more useful to them. We used the following methods as a way to achieve these goals:

1. Research current monitoring systems and best practices.
2. Determine key metrics in monitoring the health of a distributed system.
3. Configure a prototype monitoring system.
4. Design a monitoring dashboard to display distributed system metrics, including:
   a. Alerts and configurable thresholds for when they should fire.
   b. Simple, “quick-glance” visualizations that give an overview of system health.
5. Test the prototype with prospective users and refine dashboard based on feedback.

Following these methods, we researched industry standard monitoring systems such as Splunk and Datadog. We identified the technologies Company uses within their distributed orchestration system, then used industry standards and the Rate, Errors, and Duration (RED) and Utilization, Saturation, and Errors (USE) techniques to determine which metrics to include in our monitoring platform. We built a three-node cluster of virtual machines and associated software packages to simulate data transfer within Company’s distributed system. We configured another virtual machine to monitor this cluster and scrape each software service’s metrics. With these
metrics at hand, we designed and implemented a prototype monitoring dashboard to visualize the system’s health. This dashboard included an intuitive “high-level” overview page, with additional “drill-down” pages with more details about individual software services.

We then tested our dashboard with devops engineers at Company and refined the prototype’s appearance and features based on their feedback. The testing sessions revealed that the design of our dashboard was effective in abstracting individual subsystem issues and show their impact on the overall system. However, engineers did not have the ability to see individual system issues and alerts within the Grafana dashboard, and the process for configuring these alerts within Prometheus was time-consuming and limited in functionality.

This report discusses our collective work as follows: Chapters 2 and 3 cover related technical topics and terminology; Chapters 4 and 5 present our methods and implementation processes; Chapter 6 contains our testing results; Chapter 7 includes our conclusions and potential future work.
2. **Background**

This chapter explores the concepts and technologies used or referenced in this project. Section 2.1 discusses distributed computing systems. Sections 2.2 and 2.3 discuss metrics and implementation, and the history and importance of monitoring systems. Section 2.4 introduces our technology stack, including cloud computing platforms, monitoring systems, and distributed processing services.

2.1. **Distributed Orchestration and Distributed Platform Manager**

Distributed orchestration platforms are quickly growing in popularity among software engineering organizations. The concept of orchestration refers to the automation of processes that handle the configuration, coordination, and management of a complete computer system (The Kubernetes Authors, 2019). The distributed nature of these platforms relies on multiple software components on multiple virtual or physical computing systems, optimized for scalability and redundancy. In distributed computing, scalability implies that the entire system can be expanded by adding additional computing power—either in the form of more power to the same machines (called vertical scalability) or more machines to operate in parallel with the existing system (called horizontal scalability). A redundant distributed computing system increases overall fault-tolerance; if one or two machines within the system fail, others exist to pick up the workload. (IBM, 2019)

Systems running on top of distributed orchestration platforms benefit from the simplification of many operations included in a standard software engineering stack. These include automated infrastructure for code compilation, code testing, deployment, configuring microservices, and managing cross-platform compatibility (The Kubernetes Authors, 2019).
Distributed Platform Manager, a project that Company is currently developing, aims to build upon current distributed orchestration platforms by providing developers with additional pre-configured tools that they can run their software on. Distributed Platform Manager can be deployed on any platform, meaning the Company software stack can run on a variety of hardware systems without requiring extensive reconfiguration. Distributed Platform Manager aims to provide a common set of features across all Company devices, including streaming telemetry data, a common configuration system, container services, databases, logging, and more. It also hosts a common set of operations processes, including provisioning, deployment, backups, upgrades, rollbacks, disaster recovery, fault tolerance, load balancing, and API gateways. Distributed Platform Manager uses Kubernetes as its core distributed orchestration platform, and includes application services like etcd, ZooKeeper, Kafka, and more. The importance and use of these services will be discussed later in this chapter.

2.2. Metrics and Instrumentation

Metrics are a numeric representation of a piece of information about a particular process or activity running in a computer system. A single metric can show what a specific process or activity is doing, or how it’s running. Figure 1 includes an example of metrics that track hardware resource usage by the operating system. These metrics have an identifier on the left and the associated value on the right. Each group of metrics have a descriptive comment above, denoted with the pound sign (#). A collection of metrics can provide insight on the processes and activities of the entire system. Instrumentation is the process of capturing and providing access to metrics. Instrumentation tools interface with the computer system to acquire metrics, then make them available for use either through an API or a simple web server. Metrics are stateless, so
metrics processing and storage requirements do not increase significantly with fluctuations in load, unlike log generation.

Metrics are well suited for mathematical transformations like aggregation, sampling and summarization. This makes it possible to use them to calculate and report the overall health of a computer system. Metrics can be collected over time and stored in a time-series database.

```
node_cpu_seconds_total{cpu="l",mode="irq"} 0
node_cpu_seconds_total{cpu="l",mode="nice"} 2.65
node_cpu_seconds_total{cpu="l",mode="softirq"} 9.51
node_cpu_seconds_total{cpu="l",mode="steal"} 142.38
node_cpu_seconds_total{cpu="l",mode="system"} 494.7
node_cpu_seconds_total{cpu="l",mode="user"} 2631.42
# HELP node_disk_io_now The number of I/Os currently in progress.
# TYPE node_disk_io_now gauge
node_disk_io_now{device="xvda"} 0
# HELP node_disk_io_time_seconds_total Total seconds spent doing I/Os.
# TYPE node_disk_io_time_seconds_total counter
node_disk_io_time_seconds_total{device="xvda"} 74.752
# HELP node_disk_io_time_seconds_total The weighted # of seconds spent doing I/Os.
# TYPE node_disk_io_time_seconds_total counter
node_disk_io_time_weighted_seconds_total{device="xvda"} 1540.808
# HELP node_disk_read_bytes_total The total number of bytes read successfully.
# TYPE node_disk_read_bytes_total counter
node_disk_read_bytes_total{device="xvda"} 2.34228736e+08
# HELP node_disk_read_time_seconds_total The total number of seconds spent by all reads.
# TYPE node_disk_read_time_seconds_total counter
node_disk_read_time_seconds_total{device="xvda"} 5.5760000000000005
# HELP node_disk_reads_completed_total The total number of reads completed successfully.
# TYPE node_disk_reads_completed_total counter
node_disk_reads_completed_total{device="xvda"} 13858
```

**Figure 1**: Example of metrics that show system resource usage (Prometheus, 2014).

2.3. Monitoring

As software becomes increasingly focused on independant services and connected applications, understanding the state of software infrastructure and systems has become an essential part of ensuring project stability and reliability. Monitoring is the process of aggregating and analyzing metrics to provide an overview of system behavior. Monitoring systems collect metrics and other information from application components and build visualizations with them to report health and performance to the development and/or operations
teams. A robust monitoring system can simplify the development operations processes immensely by making it easier for engineers to tell which parts of the application are working properly and which are not. (Ellingwood, 2017)

2.3.1. Overview

Monitoring systems consist of three core operations: metrics, monitoring, and alerting. Examples of metrics include basic operating system components like CPU load, memory utilization, and network speed, or application-specific details like API response times and system errors. A monitoring system can track metrics over time—also called time series metrics—to understand trends, identify underlying system issues, and measure changes in performance and consumption (Ellingwood, 2017).

Examples of monitoring include a dial showing average CPU load, charts that show available memory or network utilization over time, a heatmap showing API endpoints and their associated response times, or a panel showing a count of system errors. Administrators can use monitoring to identify incidents and issues within software systems and infrastructure. For example, if the monitoring system shows high response times for certain API endpoints, the administrator can quickly see that there is a problem and move to identify the issues with those endpoints (Sridharan, 2017).

Alerting is a component of modern monitoring systems that enables administrators to configure automatic responses to changes in metric values. An alerting system could perform any number of specified actions based on its configuration. Administrators could configure the alerting system to automatically add more virtual memory to a system if the ‘available memory’ metric gets too low. They could also set up an alert that sends them an urgent message if the
system error rate exceeds an acceptable limit. A strong alerting system can save system administrators and development operations engineers time, as it prevents the need to actively monitor their applications—they will just receive alerts when something needs attention (Sridharan, 2017).

### 2.3.2. The History of Monitoring

Since the early 1990s, fundamental monitoring commands and tools (such as top, vmstat, fuser, and syslog) became a standard part of both Linux and Unix (Churchman, 2018). Most desktop operating systems included real-time monitoring, then added graphic monitoring tools in the late ‘90s. Desktop monitoring tools during this time could only monitor single systems and individual users. These monitoring systems relied on system logs to source and store metric data. The ‘80s and ‘90s also saw the development of network monitoring tools such as nmon, MTRG, and Big Brother (Churchman, 2018). Unlike simple desktop monitoring tools, these network monitoring systems kept track of multiple users’ activities and the performance of the network.

The Round Robin Database (RRD) for metric data storage was created in 1997 (Brazil, 2018), and with it came a shift to storing metrics as time series data rather than in logs. This change brought notable performance improvements to analyzing system health. The RRD became the basis for other innovative tools like Smokeping (Smoke Ping, 2019) and Graphite (Graphite, 2019). The growth of the Internet in the early 21st century brought a need to monitor websites and Internet-based services. This eventually lead to the development of tools such as Nagios, a software application that monitors systems, networks, and other technical infrastructure. These tools operate by regularly executing scripts that are referred to as “checks.” When a check fails, the script returns a nonzero exit code and an alert is sent out. A combination
or variant of both Nagios and Graphite was the dominant solution in computer monitoring before the recent shift to cloud computing (Brazil, 2019).

With the rapid expansion of the Internet and emerging interest in cloud computing technology, demand increased for more powerful “cloud native” monitoring systems. Modern monitoring tools put an increased focus on better scalability and performance in the context of multi-user, distributed, and virtualized infrastructure. Monitoring tools now do more than just record data; they provide the ability to analyze, aggregate, filter and then visualize information for rapid understanding and action (Churchman, 2018).

2.3.3. Cloud Native Monitoring

The procedures used to develop and deploy software applications has changed drastically in the past few years. The introduction of concepts like microservices, containerization, distributed orchestration, cloud computing/serverless architecture, and more have transformed the way we design, deploy, and manage software. However, as architecture gains complexity, the number of potential points of failure increases. Traditional monitoring cannot keep up with these expanding, complex architectures (Docker, Kubernetes, etc) (Sridharan C., 2017). Many modern applications are deployed at-scale to multiple server clusters across numerous availability zones, with servers being constantly added and removed from clusters. Additionally, high-traffic systems generate log files so quickly that it would be impossible for humans to keep up. In order to deploy and maintain these large-scale cloud-based systems successfully, engineers need to have a strong understanding of how the services and infrastructure are behaving; cloud native monitoring can provide this insight.

Cloud native monitoring systems use a time series database model to collect and correlate
changes in metrics over time. The methodologies behind cloud native monitoring call for an ease in integration between different tools (R. Scott, 2018). These time series databases can aggregate data from multiple sources, allowing development operations engineers insight into their entire tech stack at once—rather than having separate systems for individual parts of it. These aggregated metrics can provide significant insight into how the application is running and help expose any potential problems before they manifest.

2.3.4. Time Series Databases

A Time Series Database (TSDB) is a database optimized for data recorded over a period of time. Time series data are simply a set of numeric values, each paired with a timestamp and usually recorded over a set interval of time. Some examples of time series data include server metrics, network data, sensor data, event occurrences, clicks, trades in a market, etc. Historically, one of the major drawbacks that time series data had was its lack of support for tags and labels. The barebones nature of this data structure made filtering and analyzing the data more difficult. However, modern monitoring systems, including Prometheus, supports labels as an additional key-value pair with the metrics (Influxdata, 2019).

2.3.5. Whitebox vs. Blackbox Monitoring

There are two common classifications of monitoring methodologies: white box and black box. The core distinction is how much access the monitoring system (or tester) has to its target. In white box monitoring, the tester has access to the internal statistics of the system, including things like hardware usage, filesystem logs, unit testing, code-based failures, etc. In black box monitoring, the tester only has access to the application itself, and thus can only examine externally visible behavior. Since black box monitoring involves no knowledge of the system’s
internal structure, testing focuses on application behavior like response latency, service availability, responses to user interaction (use case testing). Generally speaking, development operations engineers are responsible for black box monitoring and software engineers are responsible for white box monitoring (SoftwareTestingFundamentals, 2019)

This paper puts a greater focus on the white box monitoring technique and how it applies to Company’ Distributed Platform Manager system.

2.4. Technology Stack

This section introduces the technologies used in our project. Section 2.4.1 covers Amazon Web Services and cloud computing. Section 2.4.2 explores Prometheus, metrics scraping, and Prometheus Node Exporter. Section 2.4.3 discusses Grafana and data visualization. Sections 2.4.4 -2.4.7 cover the subset of application services used in Distributed Platform Manager that we’ll be monitoring.

2.4.1. Amazon Web Services

Amazon Web Services (AWS) is a secure cloud services platform, offering compute power, database storage, content delivery and other functionality to help businesses scale and grow their technology operations (Amazon Linux 2, 2019). AWS offers many different services, but our project only uses their Amazon Elastic Compute Cloud (EC2) platform. Amazon EC2 provides secure, resizable, cloud-based virtual machines. It is designed to make large-scale cloud computing easier for developers (Amazon Linux 2, 2019).

We use four EC2 virtual machines (or “instances”) for our project. These instances host our monitoring software system and our data processing cluster..
2.4.2. Prometheus

Prometheus is an open-source metrics-based monitoring system and alerting toolkit (Prometheus, 2019). It includes a data model for labelled time series data and PromQL, a query language that allows users to analyze application and infrastructure performance. Company plans on using Prometheus to scrape metrics from all systems in their Distributed Platform Manager platform.

Prometheus scrapes and collects metrics from targeted services called “jobs.” Prometheus makes all scraped metric data available to external services via an API. This API can be used to request data or evaluate PromQL queries; services like Grafana use the results from these queries in the visualizations within user-created dashboards. Figure 2 illustrates the Prometheus architecture and ecosystem components. The diagram shows that metrics are pulled from instrumented jobs to Prometheus through proper exporters. Prometheus then stores all scraped samples locally and aggregate these data into a time series database. Grafana or other API consumers can be used to visualize the collected data. Additionally, other rules and queries can be applied to Prometheus to trigger and send out alerts via its Alertmanager.
2.4.3. **Node Exporter**

Prometheus Node Exporter is a hardware and operating system instrumentation tool. It collects CPU, disk, memory, network, filesystem, and other statistics—then exposes them as scrapable metrics via an HTTP server. These metrics are useful in evaluating system load on an endpoint in a network or node in a cluster (Prometheus, 2014).

2.4.4. **Grafana**

Grafana is an open-source metrics dashboard and graph composer; it allows developers to query, visualize, and send alerts based on specified conditions (Grafana Labs, 2019). Company is looking to use Grafana to create visualizations and dashboards from the Prometheus time series databases. Grafana runs as a web application and supports multiple data sources such as Graphite, InfluxDB, or OpenSB, but most implementations of Grafana use Prometheus as a data
source (Grafana Labs, 2019). Grafana can be used to visualize system metrics like CPU load, memory availability, error signals, and API endpoint latency with customizable tables and graphs. Available visualizations also include heat maps, histograms, pie charts, and measurement dials. Grafana also allows developers to import community-developed dashboards from a large community-curated repository.

Grafana charts are highly-customizable: developers can choose the color palette, label size, label location, grid display units, chart size, chart location, and many other options. The built-in Grafana alerting system allows users to forward system alerts to a third party service like Slack, email, or a custom webhook (Logz.io, 2018). Figure 3 shows an example dashboard that contains three panels containing graphs. The graphs show the customizability in the dashboard, by aggregating data from multiple sources and comparing/contrasting one metric with another metric. The dashboard also contains eight Singlestat panels which are all customized with a color coding, indicating to the user how well each metric is performing.

Figure 3: Example of Grafana monitoring dashboard (Grafana Docs, 2019).
2.4.5. **ZooKeeper/Kafka**

Apache Kafka is a distributed streaming platform that can be used to publish, subscribe to, and store streams of records. Kafka can also transform and analyze these streams of records as they occur. It is commonly used to build real-time streaming data pipelines or build a real-time applications to process streaming data and is one of the core services used in Company’s Distributed Platform Manager project.

ZooKeeper is a centralized hierarchical key-value store service for distributed systems. It is used to provide a distributed configuration service, synchronization service, and naming registry for large distributed systems. Zookeeper helps manage and coordinate tasks, state management, and configuration across a distributed Kafka system (Apache Kafka, 2019).

2.4.6. **JMX Exporter**

Similar to the Prometheus Node Exporter, JMX Exporter is a Java-based instrumentation tool. It runs as a Java Agent on top of another Java application—in this case, ZooKeeper and Kafka—and collects resource utilization statistics represented by the Managed Beans (or mBeans) within the Java Management Extensions (JMX) service. JMX Exporter then exposes these statistics, including Java Virtual Machine (JVM) information, as metrics on an HTTP server (Prometheus, 2014). These metrics are useful in evaluating the efficiency of the system’s JVM and troubleshooting any JVM-related system issues.

2.4.7. **Prometheus Alert Manager**

The Alert Manager handles alerts sent by Prometheus server and notifies end users (Prometheus Docs, 2014). It is in charge of inhibition, grouping, and routing alerts to the correct
receiver integration. Grouping is used to group similar alerts into one single notification in case of large issues and many system fails at once. Inhibition suppresses certain alerts while others are firing. Alert Manager also has the ability to mute alerts for a given amount of time, or silencing.

2.4.8. etcd

Etcd (stylized as “etcd”) is a distributed key-value storage system that focuses on reliably maintaining data within distributed systems. Company’s Distributed Platform Manager project uses etcd to maintain a common set of configurations across each of its nodes. An etcd cluster contains an odd number of nodes and uses a consensus algorithm to determine how key-value pairs are stored throughout the system. etcd maintains simplicity through its gRPC API, security with TLS encryption, speed (benchmarked at 10,000 writes/sec), and reliability through its use of the Raft consensus algorithm.

In the Raft consensus algorithm, servers can fill the roles of either a “follower,” “candidate,” or “leader.” All nodes are initialized to a follower state. If the leader is not present, any one of the follower machines can become a candidate and enter the voting process to elect a new leader. If the majority of the machines respond with a vote, the candidate becomes a leader and all requests to the cluster go through the Leader. Thus, even if the majority of servers fail, the cluster can still vote a new leader and process requests.
3. Related Work

Understanding industry-standard monitoring systems would help us better define the metrics we should include in our system and learn good dashboard design practices. Therefore, we looked at the most popular monitoring systems available on the market, with the goal of building a user-friendly monitoring dashboard with efficient alerting system for Company. The two most popular monitoring software services and platforms on the market are Splunk and Datadog. We describe these two services and their influence on our project in sections 3.1 and 3.2.

3.1. Splunk

Splunk was founded in 2003 as a software platform and service to search, monitor, analyze and visualize machine-generated log data (Splunk Cloud, 2019). Since its initial release, Splunk has grown to be one of the most frequently used monitoring tools at many large companies; their customers include 90 of the Fortune 100 companies. Splunk provides many different services, including machine learning toolkits and a query language called the Splunk Search Processing Language (SPL). Based on the scope of our project, we only focused on Splunk’s monitoring, alerting, visualization and dashboards features.

Splunk uses scheduled searches to process continuous metric data and provide real-time monitoring features. A scheduled search runs on a specific pre-set interval and keeps dashboard visualizations updated, which in turn keeps users informed about the current system health. Figure 4 shows a system health score in the top row of a Splunk dashboard. In Splunk, a health score ranges from 0-100 and indicates how a system is performing. It is calculated from all key performance indicators in the system and its status updates once every minute (Martin, 2017).
This featureset influenced our decision to create an automatically-updated single-stat score that reflects the overall health of the clusters. This score contains subcomponents representing each core component in the system, giving the user a holistic overview within a single panel.

![Figure 4: Splunk dashboard with service health score (Splunk Docs, 2019).](image)

Splunk also provides customizable dashboards and data visualizations. This includes the Trellis Layout Visualization, which has the ability to split a search result or multi-component graph into several similar visualizations. This can help users perform in-depth analysis of smaller components within the system. For example, both Figure 5a and 5b show an analyst for an online retailer searching for customer actions by product category. The standard layout in Figure 5a shows a column chart with customer actions across all product categories, whereas the Trellis layout in Figure 5b splits the visualization into separate column charts for each customer action category (Splunk docs, 2019). Grafana does not provide this feature as part of their dashboards, but this concept served as a foundation in the implementation of detailed “drill-down”
components within our dashboards; upon clicking on one of the panels, users are redirected to a new dashboard with more detailed information about that panel.

![Figure 5a: Standard Layout in Splunk (Splunk Docs, 2019)](image)

![Figure 5b: Trellis Layout in Splunk (Splunk Docs, 2019).](image)

### 3.2. Datadog

Established in 2010, Datadog is a software company that provides a Software as a Service (SaaS), monitoring platform for cloud applications (Crunchbase, 2019). Datadog works with customers to bring data from different servers, databases, and other systems together to establish a meaningful view of their infrastructure. Datadog has become a leader in the field of monitoring and analytics; they work with companies such as Airbnb, Jenkins, Evernote, and Buzzfeed, (Datadog, 2019). Datadog has developed many customizable solutions, such as their
container map product that visualizes every container across a Kubernetes system, and other
customized solutions that specifically cater to commonly used software systems.

Datadog maintains a blog that discusses their monitoring methodologies and provides
breakdowns of their metrics and alerting decisions (Mouzakitis, 2016). In these blog posts,
developers detail how to monitor specific systems, what metrics are important, and what
activities should have alerts. These posts provided us with insight into industry “best practices”
and influenced how we implemented monitoring for each of the technologies within our cluster.
For example, when deciding which metrics to record from Kafka, we referred to Datadog’s post
on Monitoring Kafka Performance Metrics, which provided explanations for why certain metrics
are more important in determining system performance than others (Miller, 2018).
4. Methodology

The objective of this project was to explore monitoring systems and development operations procedures, then build a prototype monitoring system that supports part of Company’s Distributed Platform Manager system. In pursuit of this objective, we completed the following steps:

1. Gather project requirements.
2. Determine key metrics in monitoring distributed system health.
3. Design and configure a prototype monitoring system.
4. Test the prototype dashboard with prospective users.
5. Refine the prototype dashboard based on feedback.

4.1. Gathering Project Requirements

We had three key project stakeholders: Distributed Platform Manager project management, the Distributed Platform Manager development team, and department management. Each stakeholder had a different focus within the project’s scope; project management was most interested in the development process and defining overall project goals, the development team was most interested in ensuring our dashboard conformed to existing development standards, and department management was most interested in the outcome—how the dashboard functioned and what the user experience was like.

We met with project management most frequently. In these meetings, we steadily refined our understanding of the Distributed Platform Manager technology stack, how our project relates to Distributed Platform Manager, and the project’s scope. Our project started as a general monitoring dashboard for all of Distributed Platform Manager’s components and services.
However, this scope had timeframe concerns since gaining access to the Distributed Platform Manager environment, documentation, and data might not have been possible within our timeline. Additionally, some of the technologies within the Distributed Platform Manager project had a steep learning curve and would require significant time to understand and integrate into our dashboard.

We then refined our project scope to focus on building a monitoring dashboard for certain systems and services within the Distributed Platform Manager project environment. With this approach, we would still be interacting with Distributed Platform Manager system load data—just not for the services that had a steep learning curve. However, project management became increasingly concerned that we would not be able to get access to any of the Distributed Platform Manager systems or data within our timeline. To circumvent the need for access to the Distributed Platform Manager platform, we selected certain services from the project’s technology stack and deployed them on custom AWS instances. This allowed us to simulate activity within an Distributed Platform Manager cluster and build a dashboard to monitor those services; our findings could then be used to build a similar system to interact directly with the Distributed Platform Manager system at a later date. We settled on configuring and monitoring the following services: etcd, ZooKeeper, Kafka, Prometheus, Grafana, and Node Exporter. These were chosen due to both ease of learning and their prominence within the Distributed Platform Manager project.

We did not have the opportunity to meet with department management, but their perspective and expectations were relayed through project management. The major user-experience focus was around simplicity; they wanted something similar to an airplane
dashboard—where pilots can get a quick overview of how every system is running at a quick glance, but can also dig deeper into each system’s operations if needed. Figure 6 below shows an example of an airplane dashboard provided to us by department management. It shows the flow of fluid through an airplane’s two hydraulic systems; if one of the systems at the top fails, every system below is affected and the dashboard reflects the reduced functionality.

![Diagram of airplane dashboard](image)

**Figure 6:** Example of an airplane dashboard from department management.

The development team’s standards requirements touched on how they have built and saved Grafana dashboards within the Distributed Platform Manager project. Many of their current dashboards reference official (or unofficial but widely-used) dashboards listed on the
Grafana dashboard repository. The team recommended we follow a similar process when we built ours, if possible. The development team also requested that we export our dashboards using their Distributed Platform Manager “Helm Chart” format. They use Helm to manage the Distributed Platform Manager service packages within Kubernetes. However, deploying our dashboards within Distributed Platform Manager was not in our project scope, so we were not required to adhere to those standards.

4.2. Determining Key Metrics in System Health

Before we could design and build a prototype monitoring dashboard, we first had to understand the services used in our project. This included learning how and what each service exports for metrics. We focused on the following services: etcd, ZooKeeper, Kafka, Prometheus, Grafana, and Node Exporter. In our initial research, we found that each of these services exports hundreds—and in some cases, thousands—of system metrics. Further, some of these metrics were more significant indicators of system health than others, and some could not be used to determine system health at all. To establish which metrics to use in our monitoring dashboard, we listed every metric each service produced in a spreadsheet and labelled them as “useful,” “potentially useful,” or “not useful.” We found that the number of “not useful” metrics greatly outnumbered the “useful” or “potentially useful” metrics. We also labeled the “useful” and “potentially useful” metrics with details on how they could be used in our dashboard.

etcd automatically exports its metrics via an HTTP endpoint. The etcd development team also details the most important metrics in its official documentation, along with each metric’s purpose and use (Etc.io, 2013). We cross-referenced the documented metrics with the ones in our spreadsheet and labelled them accordingly.
Kafka relies on external instrumentation services to export its metrics, the most supported one being JMX Exporter (Apache Kafka, 2015). The Kafka official documentation lists every metric that JMX Exporter provides access to, but provides no distinction of importance between any of them beyond simple categorization. Since JMX Exporter provides access to several thousand metrics, this categorization was not particularly useful—we would still have to read through all of them. We found a blog series that details the key Kafka metrics to use in a monitoring system, published by Datadog, one of the monitoring software industry leaders (Mouzakitis, 2016). This series provided a much more succinct explanation of which metrics were useful, along with objective reasoning for each metric.

ZooKeeper operates similarly to Kafka; it relies on JMX Exporter to export its metrics and the official documentation provides no distinction of importance between metrics. A document produced by Server Density broke down important ZooKeeper metrics and monitoring methodology. Server Density is a Monitoring Software-as-a-Service company owned by cloud services conglomerate StackPath (Mytton, 2016). It explained which metrics were important indicators of ZooKeeper’s system health and how they should be integrated with a monitoring system.

Prometheus, Grafana, and Node Exporter all export their own metrics via an HTTP endpoint. Our research failed to find official documentation explaining which metrics represent critical system functions and needs. To determine which metrics were most important, we used the USE (Utilization, Saturation, and Errors) and RED (Rate, Errors, and Duration) methodologies in conjunction with the basic metric descriptions provided within each service’s exports (Mushero, 2017). The USE methodology focuses on system resources and internal
metrics, whereas the RED methodology centers around metrics describing work the service does. Taking all metrics observed in a superset of both methodologies resulted in a highly descriptive overview of the service and how it performs.

4.3. Designing and Configuring a Prototype Monitoring Dashboard

After establishing our project requirements and researching important system metrics, we designed a prototype monitoring dashboard. Based on the department management’s request for a dashboard where users could check the overall system status “at a glance,” we evaluated potential approaches for aggregating the status of subsystems into a larger overall system. After that, we combined our system metrics research with the hardware specifications of our AWS instances to determine effective alerting thresholds. We then created a basic dashboard design based off of the department management’s inspiration.

4.3.1. Determining an Aggregation Process

A score-based approach would be an effective way to aggregate subsystem statuses into an overall system status. In this approach, each subsystem (an individual service like etcd, Kafka, or Node Exporter) would receive a “health score” based on a running evaluation of its metrics. These scores would all roll-up into a larger categorizations subsystems if relevant, and eventually to an overall system health score. If a subsystem were to encounter any issues—like too much resource usage or high response latency—its health score would lower, in turn lowering the overall system health score. We chose to put the score on a 0-to-100 point scale, where a zero would indicate complete system failure and a 100 indicates the system is performing optimally.

There were several different approaches we could take to calculate these system health scores, all of them involving taking measurements of the current state of each subsystem and
incorporating any system-wide alerts to determine an overall system health score. One approach would simply take an average of the state of all subsystem metrics to determine a score for the entire subsystem. This subsystem score is then included in the overall system average. This approach would require a taxonomization of each metric measurement—basically ranking all possible values of each metric on a one-to-five scale—and then aggregating the classification of each metric’s current status into its subsystem score. This process would be intensive for subsystems with large sets of metrics or complex inter-system dependencies; it would also prove challenging to apply to scalable systems, as the taxonomizations would have to be recalculated each time the system is scaled. This approach also introduced challenges in adequately representing systems that have entered a critical or non-operational state; if all but one subsystem are operating perfectly, then the average score across all subsystems might indicate no significant problems within the overall system, despite one of them being completely non-functioning.

A slightly modified approach that addresses this shortcoming would weight the overall score to disproportionately represent individual subsystem scores that have entered a critical or non-functioning state, letting the overall system score reflect both critical and non-critical system states. An example of this is as follows: if one subsystem is non-functioning and is classified as having a score of 0/100, then the overall system score is represented as 0/100. This can also be described as a system where the overall score mirrors the lowest score of its subsystems. This approach still requires the taxonomization of each individual metric, however, and thus incurs the some of the same challenges as the original approach.

An alternative approach includes representing the score as a direct indicator of how many system alerts are currently active. A system alert, triggered by activity captured in the monitoring
process, indicates an occurrence that requires external action. Naturally, things that do not require action should not require a “warning” or pre-alert, as those can be ignored until action must be taken. Thus, the state of any given subsystem within the overall system is either actionable, or it is not. If we form subsystem scores based on the number of respective system alerts that are currently active, then the taxonimization process simplifies significantly; we only need to identify the points at which each subsystem demands direct action from an administrator or engineer, rather than five separate classifications for each metric. However, this approach reduces granularity within overall system—it could be difficult to differentiate between a system that is non-functional and one that is just seeing high load based on scores alone.

Following our research and discussion, we devised a formula to calculate the health score of any given subsystem within our distributed system. This is represented below in Equation 1:

\[
Health\ score = 100 - \left( \frac{\sum (number\ of\ active\ alerts)}{\sum (total\ number\ of\ alerts)} \times 100 \right)
\]

Equation 1: Formula to calculate system’s health score

The ratio of active alerts (indicating an actionable issue within the subsystem) to total possible alerts is the core of the equation. If zero alerts are currently active, the subsystem’s score is 100; similarly, if all alerts are active, its score is 0. We can calculate this score for each subsystem in the overall system, then find the average across all scores to determine an overall system score. If necessary, the average scores can be weighted based on varying importance between the subsystems.

4.3.2. Choosing Alerts and Thresholds

We can improve the granularity of our aggregation method by setting multiple alerts for each system metric—one that indicates a degraded system state, and one that indicates a
complete system failure. The “degraded” alert triggers when there is an actionable occurrence that is not a system failure, like unusually high system load or low resource availability. The complete system failure alert will trigger during conditions where the system is not capable of handling any traffic or work. In a scenario where a system is running properly but has actionable conditions based on system load, the system score will reflect a medium state of alert. If a subsystem fails entirely, the all alerts will be triggered and the system score will reflect a maximum state of alert.

Setting the alerts for each subsystem was a three step process. First, we identified the metrics we wanted to monitor. For each metric, we then determined the condition in which that metric would indicate a state of failure. Finally, we determined the condition in which that metric would indicate a degraded subsystem state.

The process for choosing these alert conditions (thresholds) varied between subsystems and types of metrics. The metrics for some subsystems, like Node Exporter, were tied directly to our cluster’s hardware specifications. The alert threshold for failure within Node Exporter was set as a lack of metric data (indicating that the cluster is not running and therefore is not streaming metric data); the threshold for a degraded state was high resource utilization including CPU utilization above 80% and remaining storage below 10%. Other services, like etcd or Kafka, had metrics that were tied to both our cluster’s hardware specifications and application-related activity. Setting alert thresholds for these metrics required a strong understanding of how each service operates and how they use system resources. We referred to the official documentation and expert blog posts to set the degraded and failure state alerts for
these services. For more details on the implementation of these alerts and thresholds, refer to Chapter 5.6 (Implementation - Dashboard configuration).

4.3.3. Designing the Prototype Monitoring Dashboard

We began the design process by hand-drawing wireframes for four potential dashboard screens (see Figure 7). These wireframes were heavily inspired by the direction of—and examples provided by—department management. They display a potential user interaction, where the user is met with an initial system overview screen, then “digs down” into the details of a specific subsystem. We reviewed these wireframes with project management to ensure our designs were headed in the right direction. The panel in the top left corner of Figure 7 shows the main overview dashboard, which contains the health scores for the overall system and all subsystems. The other three panels contain a more detailed view of each subsystem, including additional health scores, charts, and system alerts.
Upon confirmation, we proceeded to re-create the wireframes—without functionality—using the Grafana dashboard building platform. Images of the Grafana dashboard can be found in Appendix A. This design followed the same principles as the wireframe designs, just adding color indications for the system overview based on how healthy each subsystem is. Each subsystem was grouped with other similar subsystems within the overview screen, with the overall system health score shown at the top and subsystem groups and then individual subsystem health scores shown below. This layout was intended to allow the user to work their way down the screen to determine the sources of any system health problems; they would start at the top-most panel to see the overall system health, then follow any discrepancies down through the subsystem group health panels to the individual subsystem’s health score. Users could access detailed screens for each individual subsystem from the overview screen by clicking the panel.
that represented each subsystem’s health score. These detail screens contained graphs, dials, and other visualizations that indicated how the subsystem was performing and how its performance impacted the overall system health score.

Once the Grafana-based design was in place, we began implementing some system infrastructure and load simulation methods. This process included the selection and implementation of all graphs within each subsystem’s detailed screen. These decisions are discussed in Chapter 5 (Implementation).

4.4. Testing the Prototype Monitoring Dashboard

To evaluate the prototype dashboard, we set up meetings with project management and operations engineers, both of whom would be primary users of a monitoring system for the Distributed Platform Manager platform.

We chose specific-task as our moderated usability testing method. During the usability test, we sat with the users, observe and answering their questions real time. We generated two dashboard scenarios, represented in Figure 8a and 8b. The first dashboard shows critical errors happening in one subcomponent (Kafka), but everything else is working properly. The second dashboard shows lots of degraded subcomponents but none of them are critical. Using the two dashboard scenarios, we wanted to find out how users behave when alerts occur and how users react when multiple similar alerts are fired simultaneously. Users have to identify what is going wrong within the system by navigating through the drill down dashboards. During the usability testing session, we asked users to verbalize their thoughts. This gave us a better understanding of the users' experience without having to interrupt with their thought process.
**Figure 8a:** Usability testing dashboard 1.

**Figure 8b:** Usability testing dashboard 2.
5. Implementation

To build the prototype monitoring system, we first set up and configured a functioning cluster of the technologies with which our system would interact with. Our cluster included a four-node setup, where one node hosted the Prometheus and Grafana services (used to stream and visualize the cluster’s metrics), and the other three contained internetworked Kafka and etcd services. Figure 9 visualizes the design and set up of our cluster, and this chapter discusses the setup and configuration processes for it.

Figure 9: Visualization of etcd cluster and monitoring services set up.

Section 5.1 discusses the process of provisioning AWS instances for our cluster. Section 5.2 covers the process of setting up Prometheus, Grafana, and Node Exporter. Sections 5.3 and 5.4 discuss configuring ZooKeeper, Kafka, etcd, and JMX Exporter. Section 5.5 includes our process for simulating a realistic workload on our cluster of services. Section 5.6 discusses the configurations for our Grafana dashboard.
5.1. Amazon Web Services (AWS) Instance Provisioning

Our cluster is built across four AWS EC2 instances, provisioned on the US West (Northern California) availability zone. Each EC2 instance acts as an individual, distinct virtual machine. We provisioned these instances as ‘m5.2xlarge’ EC2 models, each with 8 virtual CPU cores (based on the Intel Xeon Platinum 8000 series processor), 32GB of memory, 256GB of solid-state storage, and up to 10Gbps of available network bandwidth (Amazon Linux 2, 2019). We chose the ‘m5.2xlarge’ model because it provides enough processing power, memory, and bandwidth to run load-intensive processes on Kafka and etcd. If we ever exceeded the model’s resource limits, AWS would allow us to provision larger instances and migrate our services to the new ones.

Each AWS EC2 instance provides control over operating system choice through Amazon Machine Images (AMIs). The list of AMIs includes a large variety of Linux distributions and Windows servers. We initially chose to deploy our EC2 instances with the Amazon Linux 2 AMI, as it includes AWS integration tools and systemd support (for service scheduling, required for etcd, ZooKeeper, and Kafka) out-of-the-box (Amazon EC2 M5 Instances, 2019). However, we ran into configuration issues between the version of etcd we were trying to install and the pre-installed version on Amazon Linux 2, so we switched the three instances running etcd and ZooKeeper/Kafka over to the Ubuntu 16.04 LTS AMI (which has similar features but did not include a pre-installed version of etcd).

We also had to configure security rules for each instance to ensure nodes in the cluster could communicate properly, and that we could access each service’s metrics from outside the cluster. To do this, we made a custom AWS Security Group that contained every inbound traffic
rule that each of our services needed. This amounted to exposing the following ports for any incoming requests: 22 for SSH access, 9090 for access to the Prometheus dashboard, 3000 for access to the Grafana dashboard, 9100 for access to system metrics (via Node_Exporter), 7070 for access to ZooKeeper metrics (via JMX Exporter), 7071 for access to Kafka metrics (via JMX Exporter), and 2379 for access to etcd metrics. Security Groups also give users command over which types of traffic can access certain ports; for example, we could configure the inbound traffic rules such that no external traffic could access the nodes within our cluster—thus ensuring that only our Prometheus server had direct access to the metrics they produce. This configuration would increase the security of our cluster by increasing our control over what information external parties are able to see; however, this level of security was unnecessary since this system is just a prototype and is inaccessible outside of the Company internal network.

5.2. Prometheus, Grafana, and Node Exporter Setup and Configuration

We use Prometheus to stream metric data in a time-series format and then use Grafana to visualize that data. Both services have well-documented initial setup and configuration processes, but needed additional customization to fit our setup. We installed both services on the same Amazon Linux 2 (CentOS-based) EC2 instance within our cluster. We also configured the Prometheus Node Exporter, which reports system-level metrics like CPU utilization and available memory, to run on this instance.

The initial Prometheus setup and configuration was fairly straightforward: download Prometheus, write a configuration file, and feed the configuration file to Prometheus in the startup command. The configuration file contains settings that dictate where and how frequently Prometheus looks for new metrics. We configured our Prometheus server to scrape for new
metrics every fifteen seconds, which provides a fairly granular look into each service’s activity compared to the default of one minute. We also configured a “job” for each service we’d be monitoring; these job configurations simply contained a descriptive name and a list of the “targets” (URL/port pairs) from which Prometheus would scrape the metrics (Prometheus, 2019). We initially configured only one job—for Prometheus to scrape its own metrics—but added additional jobs once we got their respective services running.

The Grafana installation process is almost entirely handled by the YUM package manager. Grafana also comes with a pre-configured systemd server setup, so the entire service can be installed and run with just two commands. The configuration process is just as straightforward; once the Grafana server is running, simply log in with the default username/password combo and it will step through the process of configuring data sources and dashboards (Grafana Labs, 2019). We pointed Grafana to our Prometheus server as its primary datasource, and it automatically connected and made our time-series metrics available for use in a dashboard.

Installing the Prometheus Node Exporter is as simple as downloading the binary and running it (Prometheus, 2014). However, we needed a way to ensure that the Node Exporter would run consistently and automatically restart in the case of a failure. To accomplish this, we wrote a custom systemd service file to handle the initial startup and ensure that the Node Exporter remained running (Prometheus, 2014).

Both Grafana and the Node Exporter automatically make all service metrics available at their default ports (3000 and 9100, respectively), so we added “job” configurations for each of
them to the Prometheus configuration file. Prometheus began automatically scraping those metrics and made them available for use in Grafana dashboards.

5.3. **ZooKeeper, Kafka, and JMX Exporter Setup and Configuration**

We installed Kafka using the instructions on its official website (Apache Kafka, 2019), which included installing Java 8 using the YUM package manager. ZooKeeper came pre-packaged with our Kafka installation. Both Kafka and ZooKeeper are run with a single terminal command each; they stop running when the users exit the terminal, though. To set up ZooKeeper and Kafka to run uninterrupted, we wrote systemd service files for each. This ensures that Kafka and ZooKeeper always run in the background and restart automatically in the case of a failure.

Kafka and ZooKeeper do not automatically export their internal metrics. To access them, we installed JMX Exporter, which ran as a Java Agent and exposed the Kafka and ZooKeeper JVM metrics via a HTTP server (Prometheus, 2014). The JMX Exporter installation process required us to write a configuration file for both Kafka and ZooKeeper, then include the JMX Exporter executable file as a Java Agent in each service’s runtime arguments. JMX Exporter makes all service metrics available at port 7070 for ZooKeeper and port 7071 for Kafka, so we added “job” configurations for each of them to the Prometheus configuration file. Prometheus began automatically scraping those metrics and made them available for use in Grafana dashboards.

5.4. **etcd Setup and Configuration**

etcd operates as a key-value storage system in our cluster, specifically for the purpose of maintaining a synchronized settings package across each node. The official etcd documentation
provides many cluster configuration details that were extraneous to our setup needs, so we followed a more focused setup and configuration guide by DevopsCube (CoreOS, 2019). Once we installed etcd on each node in our cluster, we created a systemd service to ensure the service restarted properly in the case of a system failure. The etcd settings configuration followed that of a standard cluster setup—we gave each etcd node a unique name, provided the names and URLs of the other nodes, and told each node which port to make itself available on (DevopsCube, 2018). From here, the etcd nodes individually made contact with each other and used the RAFT algorithm to negotiate internal cluster settings to determine which node became the cluster leader.

However, our etcd nodes initially could not contact each other to join the cluster despite knowing exactly where the other nodes are and how to connect to them. We discovered that the Amazon Linux 2 operating system uses its own etcd installation to handle system-level processes; this appeared to be interfering with our custom configurations and impeding each node’s ability to connect with another. To remedy this, we terminated the three AWS instances running Amazon Linux 2 and provisioned three identical new ones that ran Ubuntu 16.04 LTS instead. Following the same setup and configuration steps as before resulted in a fully-connected and functioning etcd cluster across our three nodes.

etcd automatically posts its internal metrics at port 2379, so we added a “job” configuration for it to the Prometheus configuration file, listing each node as a separate metric target. Prometheus began automatically scraping those metrics and made them available for use in Grafana dashboards.
5.5.  Simulating a Realistic Workload

Once we had configured our cluster’s infrastructure, we ensured that they worked correctly and produced meaningful metric data. We wrote three Python scripts that generate data within each instance’s operating system and the Kafka and etcd services. The services propagated this data throughout the cluster and reflected any load placed on the system in the metrics they exported. The metrics produced under load helped us pinpoint alert thresholds and see which system resources the services use more heavily.

In order to add workload to each individual instance, we created a Python script to write empty text files to a specific directory at a random pace (of one file written between a range of 0 to 60 seconds). This introduced a very basic level of system load within each instance operating system, specifically targeting disk use and CPU utilization. We simulated load in etcd using a Python library called ‘python-etcd3.’ This is a Python client for the etcd API (Kragniz, 2016), which allowed us to perform simple create, read, update, and delete (CRUD) operations on the etcd clusters. We used this library to generate random key-value pairs to the etcd database at randomized rates across all three nodes in the cluster.

Before putting any load on the Kafka cluster, we created a Producer and Consumer on each node. A Producer is a Kafka client that publishes records (messages) to a different node in the Kafka cluster. Once records are published to the Kafka cluster, a Consumer on the other node then process the published records and stores them locally (Kafka, 2015). We then created a Python script to automatically send these messages through the Producer at random intervals. The script also randomized both the size of each message and the probability that each message would get forwarded to another Kafka node. This gave each node a non-deterministic influence...
on the overall system load, producing scenarios with normal and abnormal activity on each cluster.

5.6. Dashboard Configuration

The Grafana visualization platform allows users to build dashboards with three core panels: “singlestats,” graphs, and tables. “Singlestats” are a basic panel containing only one statistic and an optional simple visualization (like a gauge or unlabeled line graph in the background). Graphs can only be time-series (line charts where the x-axis is always time) or histograms. Each of these components get their data from Prometheus via PromQL queries; for each panel in our dashboards, we specified the metrics we wanted to pull, the range of time we wanted to pull from, and any filters we wanted to apply on that set of data (like filtering CPU usage by a specific process). Graph and table panels can support multiple queries which allows us to represent data from multiple sources in the same visualization.

Our dashboard included a single “main overview” screen and detailed “drill down” screens for each service/subsystem in our cluster. We built the overview screen with singlestat panels representing the health score of the overall system and each of its subsystems. The color of these singlestat panels matched the health score of the system it represented—if the score was low, the panel’s background turned red (less than 40); if the score was medium, the panel turned orange (between 40 and 80); if the score was high, the panel turned green.

Each of the subsystem panels linked to its respective “drill down” screen. This screen contained detailed visualizations based on the metrics produced by that subsystem—charts of resource utilization, process uptime, and system activity. Each chart contained individual or groups of metrics based on our research into the critical metrics for each system. Like the health
scores in the main overview screen, the metrics for each graph were pulled from Prometheus via PromQL queries.
6. Evaluation

This chapter contains the results of testing our prototype dashboard with Company engineers. Section 6.1 discusses feedback from a Principal Operations Engineer and Director of Site Reliability Engineering. Section 6.2 discusses feedback from Distributed Platform Manager project management. For both user testing sessions with the Principal Operations Engineer and Distributed Platform Manager Project manager, we used the methods and procedures as described in Chapter 4.4 (Methodology - Testing the Prototype).

6.1. Feedback from Director of Site Reliability Engineering

Dilip Sundarraj is a Principal Operations Engineer and the Director of Site Reliability Engineering with the Solutions Engineering team at Company. Dilip and his team are a target user for an Distributed Platform Manager monitoring dashboard. Dilip liked our overall design choices, especially those made for the layout of the etcd dashboard. He also suggested some improvements that would help his team identify issues in the system more effectively. He thought we should keep the design of the main dashboard as simple as possible and only include panels and alerts that are absolutely necessary to show the health of the overall system. Dilip also recommended that we clearly divide our dashboard into two sides: one showing the health of the distributed system services (such as hardware health and etcd health); the other showing the health of all application services (like ZooKeeper and Kafka). Dilip also suggested that we incorporate more advanced techniques in displaying system metrics in graphs. More specifically, he requested the use of smoothing functions (like a moving average) in visualizations that depict metrics with high volatility. Moreover, Dilip strongly suggested that we should set multiple alert
thresholds for each visualization. This would allow him and his operations engineers to distinguish between minor, major or critical alerts and prioritize their response accordingly.

Finally, Dilip charged us to carefully consider the layout of visualization panels within the service detail dashboard screens. Figure 10 shows a partial view of our prototype dashboard for Kafka service application. This figure shows Kafka’s JVM Memory usage and Average Message Per Second in the top row underneath the overall service health score. There is not necessarily a direct correlation between the number of messages sent and the amount of memory used in the JVM. However, the dashboard’s layout implies that there is a correlation between the two. This should be changed to show a clear distinction between the two metrics.

Based on Dilip’s feedback, our dashboard successfully gives users a basic overview of the system’s health and properly gives alerts when actions need to be taken. However, the dashboard needs to be extended to include insight into the root causes for each alert and a categorization of their severity. These improvements are discussed in Chapter 7.3 Future Scope.

**Figure 10**: Partial view of our prototype’s Kafka dashboard.

### 6.2. Feedback from Distributed Platform Manager Project Management

Lily Dayton is our mentor and a member of the Distributed Platform Manager management team. We met with her to reevaluate our dashboard and explore potential next steps.
In the first feedback session, Lily expressed that she was satisfied with the idea of having a single stat score to represent the health of the overall system and each of its subcomponents. She reiterated that Grafana is only a visualization tool, and that it might not include some features we needed to build a fully-functional monitoring system. Lily suggested that we rely on Prometheus to handle alerts through its “alerting rules” configurations, rather than attempting to establish alert thresholds in Grafana.

Using this approach, we were able to associate multiple severity levels with each alert and group them with other alerts for the same services. These are features we were not able to configure using the Grafana alerting system. Figure 11 shows some examples of alerting rules that we defined in a Prometheus rule.yml file. Each rule contains an alert name, alerting threshold, severity level, basic description, and summary. We are not able to view the job label (indicating which service the rule is for) on each alerting rule.

<table>
<thead>
<tr>
<th>etcd_alerting_rules</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Rule</strong></td>
</tr>
<tr>
<td>alert: etcdCheckLeader</td>
</tr>
<tr>
<td>expr: max(etcd_server_has_leader) &gt;= 0</td>
</tr>
<tr>
<td>labels:</td>
</tr>
<tr>
<td>severity: etcd_critical</td>
</tr>
<tr>
<td>annotations:</td>
</tr>
<tr>
<td>description: '({ $labels.instance }) (measured by ({ $labels.job })) no leader.'</td>
</tr>
<tr>
<td>summary: Instance {{ $labels.instance }} - no leader</td>
</tr>
<tr>
<td>alert: RPC_rate_alert</td>
</tr>
<tr>
<td>expr: sum(rate(grpc_server_started_total(grpc_type=&quot;unary&quot;)[5m])) &gt;= 0.2</td>
</tr>
<tr>
<td>labels:</td>
</tr>
<tr>
<td>severity: etcd_warning</td>
</tr>
<tr>
<td>annotations:</td>
</tr>
<tr>
<td>description: '({ $labels.instance }) (measured by ({ $labels.job })) high rate.'</td>
</tr>
<tr>
<td>summary: Instance {{ $labels.instance }} - high rate</td>
</tr>
</tbody>
</table>

**Figure 11:** Prometheus alerting rules for etcd.
We also discussed the possibility of changing Equation 1 to account for critical system alerts that indicate a complete failure. Equation 2 sets the health score of the system to 0 when any part of the system enters a critical state. Otherwise, the health score of the system is defined by Equation 1. This new formula is established in Equation 2 below:

\[
\begin{align*}
\text{If } & \text{alerting\_rule\{severity="critical"\}}: \\
\text{Health score} & = 0 \\
\text{Else:} \\
\text{Health score} & = 100 - \left( \frac{\sum (\text{number of warning alerts})}{\sum (\text{total number of alerts})} \right) \times 100
\end{align*}
\]

**Equation 2**: Revised health score calculation formula.

We were able to set a severity level for each alert threshold. However, PromQL does not support “if/else” logic, so we were not able to implement Equation 2 in our queries. In our discussion with Lily, she suggested that we clearly document this limitation and proceed with our current dashboard configuration. Additional next steps are discussed in Chapter 7 (Conclusions and Future Scope).
7. Discussion

There are some drawbacks in implementing the monitoring dashboard and alerting because of the limitations of Prometheus and Grafana. Prometheus is effective for scraping and performing simple aggregation over metrics. However, the ability to write more complex aggregations in PromQL is restricted. For example, PromQL does not provide the ability to conditionally define variables; we had to create static recording rules in Prometheus to recreate similar functionality. The same limitations arose when we attempted to define an alerting rule for Prometheus Alertmanager. These recording and alerting rules must be manually configured in a YAML file, which greatly limits PromQL’s ability to execute sophisticated query functions and affects the query accuracy and scalability. Figure 12 shows a sample of a Prometheus alerting rule. Users must define an alert name and an expression in the PromQL query format. Additionally, users can specify the duration between alerts in Prometheus using the “for” clause, set severity levels for alerts under “labels,” and add extra summary information under “annotations.” All alerting rules are defined within the same rules file, and they can be grouped with similar rules under the “groups” tag.

```
groups:
   - name: example
     rules:
        - alert: HighErrorRate
          expr: job:request_latency_seconds:mean5m{job="myjob"} > 0.5
          for: 10m
          labels:
            severity: page
            annotations:
              summary: High request latency
```

Figure 12: Sample alerting rules for Prometheus.
Unfortunately, PromQL only supports queries on the alerts that are currently being fired, even though Prometheus maintains a register of all alerting rules. This means that the monitoring system cannot determine the count of all alerting rules for each group and cannot apply Equation 1 to calculate the health score for the entire system and its subcomponents. As a temporary solution, we defined the total number of alerting rules in each group as a static variable. Unfortunately, this number must be updated manually every time new alerting rules are added to the group, making it unscalable and error-prone.

Another drawback of Prometheus alerting rules system is that there is no type nor classification requirements when defining the severity of alerts. Instead, severity can be set to any user-defined string in the rules file. Since this field is free text, any typing error, or even a change in capitalization, can result in a faulty health score query. This approach might be acceptable on smaller projects, but as it does not scale well for complex systems like Distributed Platform Manager.

Finally, Company’s dashboard requirements cannot be satisfied by Grafana alone. Grafana cannot visualize Prometheus alerts, and its internal alerting rules are not robust enough for use given Company’s enterprise requirements. Grafana is effective in turning Prometheus time series data into visually appealing graphs, but it does not support extensive customization of the display panels and alerts within dashboards. For example, the Prometheus alerting rules described above cannot be connected to Grafana alerting rules. Grafana allows users to set alerting rules independently of Prometheus, and these rules are visualized in a list, as shown an example in Figure 13. Unfortunately, these alerting rules cannot be included in Prometheus alerting rules, and have no way to indicate alert severity.
Figure 13: Grafana alerting rules visualized in a dashboard.
8. **Conclusion and Future Works**

Developing for distributed systems increases the technical complexity of software projects, and thus increases maintenance requirements. Distributed systems require development operations (devops) engineers to keep track of a large set of components and services within each software package to ensure that their systems are running properly. Monitoring software makes it easier for systems administrators and devops engineers to see how system components are functioning by automatically aggregating the status of these system components. Unfortunately, there are not many enterprise-grade monitoring solutions that are available to license—and well-established solutions have licensing fees that exceed the budget for Company and some of Company’s customers.

The goal of this project was to explore open-source software to support distributed system monitoring. Using open-source software, we successfully set up a multi-node distributed system with applications similar to those running in Company’s distributed Distributed Platform Manager system. Using Prometheus Node Exporter and JMX Exporter, we scraped and identified hundreds of potentially useful metrics for monitoring the performance of the system and its applications; Prometheus returns 125 Kafka metrics, 502 Zookeeper metrics and 1500 etcd metrics. After importing these metrics into Grafana with PromQL, we created a single overview dashboard that reflects the health of the entire distributed system, including drill-down screens to show additional details for each component.

There are three tiers of support engineers at Company. Tier one engineers are responsible for detecting any issues existing in the system. Tier two engineers are able to determine the causes for those issues. Tier three engineers know how to fix the issues based on the causes
reported by tier two engineers. Based on our usability testing session with a tier-three operations
engineer, we determined that our monitoring dashboard and alert system is best suitable for tier
one engineers and other users with less technical responsibilities. The dashboard is helpful in
determining the source of any problems within the system, but needs additional functionality to
identify and suggest follow-up actions based on the root causes of any issues.

Proceeding with this project will require additional research into alternative technologies
and approaches to data analysis. Future work includes finding an alternative to or additional
platform for Grafana. Grafana does not provide a set of features that is rich enough to handle
monitoring visualizations in an enterprise environment given the requirements of a complex
monitoring dashboard and associated alerting techniques for large distributed computer systems.
Potential approaches include finding a more feature-heavy visualization tool for Prometheus time
series data, or embedding Grafana into a web application to enable additional customization.

Other future work could include implementing the recommendations provided by
Company’s Director of Site Reliability Engineer, detailed in Chapter 6.1.1. These include
applying machine learning techniques in and simple aggregation querying to the dashboard
visualizations. The use of machine learning models could help predict the time certain system
components will fill available resources, or calculate the moving average of volatile metrics to
give more accurate insight into how the system is performing. Aggregation queries involve
deriving more complicated equations to calculate the health score for each system component
and the overall system. This can be achieved by adding weights to the alerts of each metric
depending on their importance to the performance of the system. Finally, if Company wants to
upgrade the dashboard for use by tier two and tier three devops engineers, more detailed drill-down panels and screens need to be added into the existing dashboard.

Existing Software as a Service (SaaS) products for monitoring systems, including Splunk and Datadog, provide potential platforms for monitoring distributed systems. The metrics used to monitor our prototype system were derived from blog posts and documentation of these industry-standard services. Future work could have Company investigating whether the value gained from licensing Datadog or Splunk outweighs their respective costs. Nonetheless, the use of these services is dependent on whether Company is monitoring in-house systems or providing solutions for a customer; in cases of the latter, Datadog or Splunk would have to be licensed by the customer.
9. References

Amazon. Amazon EC2 M5 Instances. Retrieved February 2, 2019, from aws.amazon.com/ec2/instance-types/m5/

Amazon. Amazon Linux 2. Retrieved February 2, 2019, from aws.amazon.com/amazon-linux-2/


Core OS. Etcfd Clustering Guide, Retrieved February 3, 2019, from coreos.com/etcfd/docs/latest/op-guide/clustering.html


Smoke Ping. About SmokePing. Retrieved February 20, 2019, oss.oetiker.ch/smokeping/


10. Appendix A

Figures A1a and A1b: The main overview screen, showing health scores for the whole system and each subsystem. A1a shows a scenario where Kafka has entered a critical state. A1b shows a scenario where the system has encountered widespread degradation.
**Figure A2**: The hardware health details screen, showing CPU load, available memory, available storage, and network utilization graphs.

**Figure A3**: The data queue overview page, showing health scores for each component of the system’s data queue software.
Figure A4: The Zookeeper health details page, containing application activity visualizations for outstanding requests, node count, average latency, and buffer sizes.

Figure A5: The Kafka health details page, containing visualizations for JVM resource usage and message/data transfer rates.
Figure A6: The etcd health details page, containing visualizations for remote procedure calls rates, data synchronization operations, and internal proposal operations.

Figure A7: The Grafana health details page, containing visualizations for resource usage, API HTTP responses, and API request latencies.
Figure A8: The Prometheus health details page, containing visualizations for uptime, error rates, scraping and reloading errors, and alert notification rates.