UNIVERSITÀ DEGLI STUDI DI PADOVA

Dipartimento di Ingegneria dell'Informazione
Master's degree in Computer Engineering

MASTER'S DEGREE THESIS

MONITORING AT HIGH SCALE FOR VERY HETEROGENEOUS DISTRIBUTED SYSTEMS

Supervisor:
Prof. Francesco Silvestri

Student:
Leonardo Bellin

Academic year 2023/2024

Graduation date 15/10/2024

Contents

Abstract **3**

1 Introduction **5**

2 Typical Monitoring Systems in the Industry **7**

2.1 Nagios . . . 7
2.1.1 Software Architecture and Functionality . . . 8
2.1.2 Plugins . . . 9
2.1.3 Strengths and Weaknesses . . . 10
2.1.3.1 Strengths . . . 10
2.1.3.2 Weaknesses . . . 11
2.2 Zabbix . . . 12

3 The Old Monitoring System at The Company **13**

3.1 Overview . . . 13
3.2 Components . . . 13
3.2.1 Prometheus . . . 13
3.2.1.1 Features . . . 14
3.2.1.2 Metrics . . . 14
3.2.1.3 Components . . . 15
3.2.1.4 Architecture . . . 15
3.2.1.5 PromQL . . . 17
3.2.1.6 Strengths and Weaknesses . . . 18
3.2.2 Grafana . . . 18
3.2.2.1 Alerting . . . 20
3.2.3 PostgreSQL . . . 21
3.2.4 Cortex . . . 21
3.2.4.1 Distributor . . . 22
3.2.4.2 Ingester . . . 23
3.2.4.3 Hash ring . . . 23
3.2.4.4 Querier . . . 23
3.2.4.5 Compactor . . . 23
3.2.4.6 Store Gateway . . . 24
3.2.4.7 Optional Services . . . 24
3.2.4.8 Challenges in Horizontal Scalability . . . 24
3.3 Architecture . . . 25
3.4 Limitations and Challenges . . . 27

4 The New Monitoring System at The Company 29
4.1 Overview 30
4.2 Architecture 30
4.3 Components 31
4.3.1 Mimir 31
4.3.2 Loki 32
4.3.2.1 Loki Stack 32
4.3.2.2 LogQL 33
4.3.3 Sentry 36
4.3.3.1 Error Monitoring 37
4.3.4 PagerDuty 39
4.3.4.1 Key Features and Strengths 40
4.4 Logging 42
4.5 Challenges 43
4.6 Ephemeral Jobs 43
4.7 Alerting 44
4.7.1 Alerting rules 44
4.7.1.1 Alert rule evaluation 45
4.7.1.2 Alerting instances 46
4.7.1.3 Notifications 47
4.7.1.4 Recording Rules 49
4.7.1.5 Architecture 49
4.8 Error Rate monitoring 49
4.8.1 SLIs 49
4.8.2 SLOs 50

5 Analysis 53
5.1 Theoretical Analysis 53
5.1.1 Evaluation Criteria 53
5.1.2 Parameters 54
5.1.2.1 Fixed parameters 54
5.1.2.2 Variables 54
5.1.3 Cost analysis 55
5.1.4 Scalability analysis 58
5.1.5 Traffic Spikes analysis 58
5.1.6 Maintenance analysis 59
5.1.7 Summary 59
5.2 Simulations 60
5.2.1 Scenario 1 60
5.2.2 Scenario 2 61
5.2.3 Scenario 3 62

6 Conclusions 63
6.1 Results 63
6.2 Future Work 63

Abstract

As the portfolio of products of a company expands, so do the number of components to keep track of, and with that, it also highly increases the complexity of the systems in place.
For product based companies it is essential to have a deep and responsive monitoring system, in order to promptly identify, respond and fix critical issues.
This thesis will present a state-of-the-art monitoring system developed for a real-world, fast-growing enterprise to deal with this situation. Given the scope of The Company, largely based on acquisitions, and the volume of transactions, it is clear that the infrastructure behind all the different products will be extremely heterogeneous and time-critical, and as such, a very flexible and efficient monitoring solution will need to be implemented.
This thesis will also highlight the main key strengths that such a model offers, demonstrating practical real-life scenarios and situations, comparing it with the old system that was in place in The Company as well as some other typical monitoring solutions that are commonly found in the tech industry. The analysis focuses on a several main aspects, such as cost, ease of setup and use, scalability and resilience to traffic spikes.

Chapter 1

Introduction

With the rapid increase of complexity in the software industry [1] and the rising demand in scalability [2], comes the need for all companies to consistently and effectively keep an eye out for issues that will inevitably happen to any system, be them caused by human mistakes, provider outage or hardware failure.
It goes without saying that in a trillion dollar industry [3], such as the software one, even small interruptions in their services cost companies billions of dollars per year [4]. That's why all the so called "big-tech" invest heavily on research and development into new, cutting-edge and always better monitoring systems to reduce the loss to a minimum [5].

This is a piece that is often overlooked by small and medium companies, as they do not see the trade-off between time spent and cost to implement such a monitoring system and the savings that it entails as a beneficial one. The aim of this study is to show that this is not the case, and provide a practical, state-of-the-art solution that is not only easily implemented by small and medium companies, but also scalable and future proof, so that the maintenance time of such system does not become a burden on the company's costs. With the goal to be as generic therefore useful for most companies, it is also designed to be flexible, easily fitting both homogeneous and heterogeneous systems, as well as both centralized and distributed ones.

This study is based on the ongoing effort to implement a better scalable monitoring system in real-world, fast-growing enterprise, which is one of the biggest software house in Italy. Throughout this study, it will be referred as *The Company*.
The business model of this company, which relies on acquisitions of new products, has led it to grow exponentially in the last 10 years, proving its very high upsides in term of growth. It, although, has certainly also shown one of its biggest downsides: the mix of technologies used proves really tough to overcome and requires years of hard work before a sort of company-wide standards is reached. Giving the now spread portfolio of products and huge money flows, it is clear that a very flexible, fast and scalable solution must be developed, in order to be fit for different technologies and prevent losses of money in case of issues.

The study will firstly give a brief overview of a typical monitoring system that can be found in any small to medium company, analyzing its strengths, weaknesses and ways of improvement. It will then give a detailed description of the old system that was in place in The

Company before the development of the new one, which will give a sense to what can be typically found in bigger companies.
The fourth chapter will focus on the architecture of the new monitoring system, explaining the structure and the working of each piece, as well as how they interconnect with one another. This part aims at giving an in-depth insight on the whole system, giving an example on how this works with microservices in The Company and what the workflow is.
The fifth chapter will try to analyze the models in both a theoretical and practical way, using practical metrics, such as cost, time to setup and scalability, highlighting the benefits and drawbacks of one over the others. It will try to support the results of the theoretical analysis with some practical experiments, based on real life scenarios.
In the conclusion, the paper will outline future research directions based on promising technologies that have emerged in recent years, which address some of the inherent issues present in current solutions.

Chapter 2

Typical Monitoring Systems in the Industry

This section aims at giving a foundational monitoring system that we can assume as the baseline for small and medium companies. This will help the analysis in chapter 5 at better giving the scale of the improvements that an ad-hoc system would provide.
Clearly it is impossible to know what each company is using and hence derive an average baseline becomes a very hideous task. Given the lack of information about this topic, which is typically under industrial NDAs, this part will describe a system that is considered the standard in the Industry.

Open-source software solutions are rapidly becoming the favourite choice in the domain of infrastructure monitoring across both small and large enterprises. Small companies can benefit from the cost-free nature of open-source tools, while large enterprises can leverage their flexibility, allowing for extensive customization to meet specific needs.
These systems dominate the market because they combine the best of both worlds: they are freely available, yet their developers often offer them as Software as a Service (SaaS) with additional features and support, making them an excellent choice even for those who want an out of the box solution. This approach combines the adaptability of open-source software with the convenience and reliability of proprietary solutions, which is way this is the preferred way for most monitoring developers. Among the hundreds of open-source projects, Zabbix and Nagios have emerged as two of the most popular options in the last twenty years.

2.1 Nagios

Although Zabbix has been gaining significant ground in popularity and represents a much broader range of applications, Nagios still largely holds the reputation of being the industry standard for monitoring IT infrastructure. Over the years it has been acknowledged numerous awards, including: Infoworld's Best of Open Source Software (BOSSIE) 2008 Award under the "Server Monitoring" category [6], the Linux Journal Reader's Choice 2009 award for "Favorite Linux Monitoring Application" from G. Jame, "Readers' Choice Awards 2009" [7], and for the 12th consecutive year, Nagios was named the Network Monitoring Application of the Year at the LinuxQuestions.org Members Choice Awards [8]. Its strong orientation toward IT operations has made it a representative of reliability and efficiency for big enterprise customers worldwide. That's why industry leaders like Apple, IBM, Sony, PayPal,

NASA, Ford, HP, UPS, and McAfee have utilized Nagios for their critical monitoring needs in the last twenty years [9].
With a long history in the industry, Nagios often becomes the main choice of organizations looking for robust and reliable monitoring solutions.
Given its status as the industry standard, and considering that this study is particularly focused on IT infrastructure monitoring, we will take Nagios as the baseline monitoring tool for our analysis. This will help to better understand its efficiency at managing and maintaining complex and heterogeneous distributed systems against other alternatives, such as the ones implemented at The Company.

From 2010 to 2015, Nagios Core has been one of the favorite open-source network monitoring solutions and today counts more than 8 million downloads from official sources.
Although it is still loved by the monitoring community, its roaring popularity started to decrease around 2016. It is believed that this loss of interest started when other more automatic and easier-to-deploy tools started to appear.

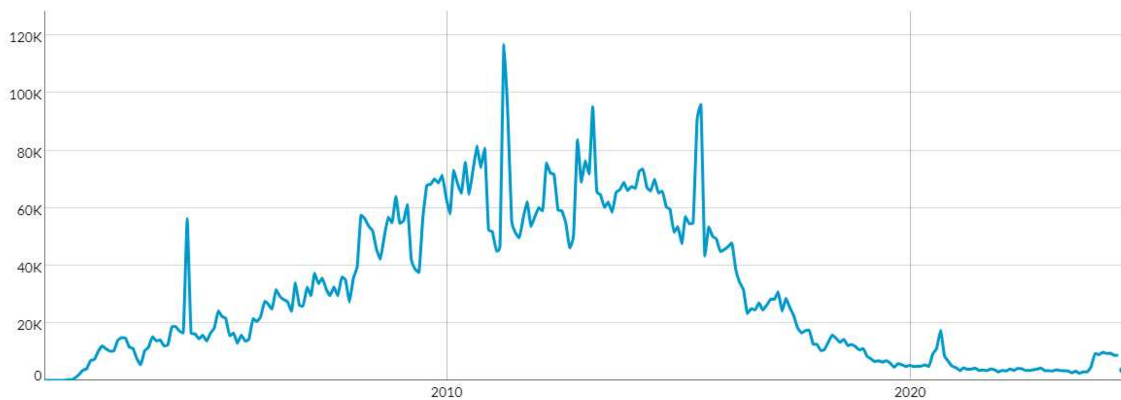


Figure 2.1: Nagios Core downloads over the years from SourceForge.net [10]

2.1.1 Software Architecture and Functionality

Nagios architecture follows the simple client-server network pattern:

- The Nagios client is the host that we want to monitor. This monitored object, which can be either local or remote, runs a Nagios agent in the background that is in charge of collecting metrics data such as CPU, memory and network usage. It then sends it back to the server for data recollection and elaboration.

- The Nagios server, on the other hand, is usually a Linux platform hosting the Nagios Core monitoring application. The Nagios Core server is responsible for scheduling, executing, processing and managing events and alerts for all the monitored objects.

The Nagios client-server communication flow works with a three steps process:

- The server periodically sends instructions to the monitored objects to execute one or more plugins and scripts. It then stores and processes the results of these instructions.

- The remote agents running the Nagios plugins gather the status requested by the server (i.e. the specific metric that the server requests) and send them back to the Nagios process scheduler. The Nagios client may get either active or passive checks: active checks are periodically requested by the Nagios server through plugins and use either TCP/IP or SNMP protocols (for example http status checks), passive checks, on the other hand, gather performance data from application servers directly without the need for the Nagios Core server to request them (for example hardware performance).
- Finally the Nagios scheduler process in the Nagios Core server updates the Graphical Web Interface (GUI) and sends notifications (or alerts) via SMS or emails back to the admin. In addition, the performance data from passive checks can be saved in a Long Term Storage (LTS) like a database to be used for graphing.

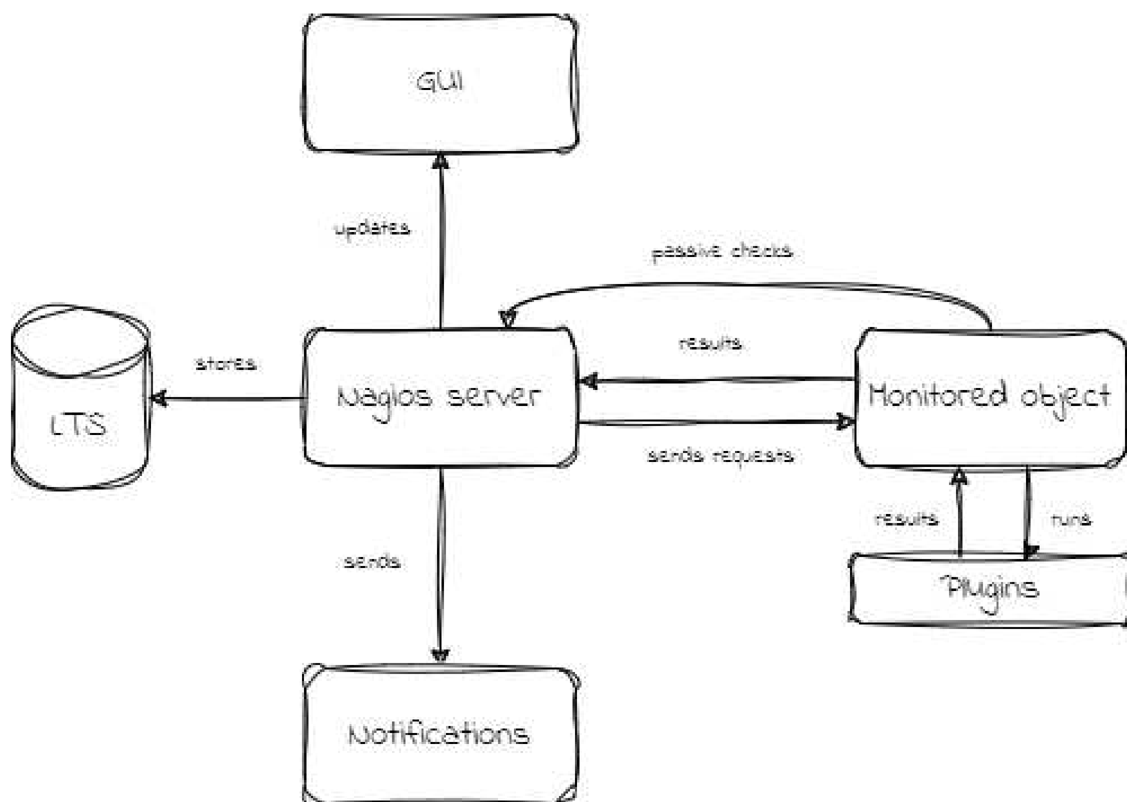


Figure 2.2: Nagios client-server flow

2.1.2 Plugins

Without plugins, Nagios Core wouldn't work. Plugins are essential extensions of Nagios Core that help monitor anything, indeed, it does not include any internal mechanisms for checking the status of hosts and services on the network by default and instead relies on external programs (plugins) to do all the dirty work.
Plugins in practice are compiled executables or scripts (Perl scripts, shell scripts, Python, PHP, Ruby, etc.) that can be run from a command line to check the status or a host or service. Nagios Core uses the results from them to determine the current status of hosts and services on the network of interest.
Nagios Core will execute a plugin whenever there is a need to check the status of a service or host. The plugin does its magic to perform the check and then simply returns the results

to Nagios Core, which will then process the results that it receives and take any further necessary actions such as running event handlers, sending out notifications, etc.

Plugins act as an abstraction layer between the monitoring logic present in the Nagios Core daemon and the actual services and hosts that are being monitored.
The upside of this type of plugin architecture is extensibility, that is, you can monitor just about anything if you can automate the process of checking something. There are already a lot of plugins that have been created in order to monitor basic resources such as processor load, disk usage, ping rates, etc.
The downside to this type of plugin architecture is the fact that Nagios Core has absolutely no idea what it is that it is monitoring. It could be anything between network traffic statistics, data error rates, room temperate, CPU voltage, fan speed, processor load, disk space, etc. Nagios Core will not understand the difference between them and the specifics of what's being monitored, it just tracks changes in the state of those resources. The separation of concerns makes it so only the plugins themselves know exactly what they're monitoring and how to perform the actual checks.

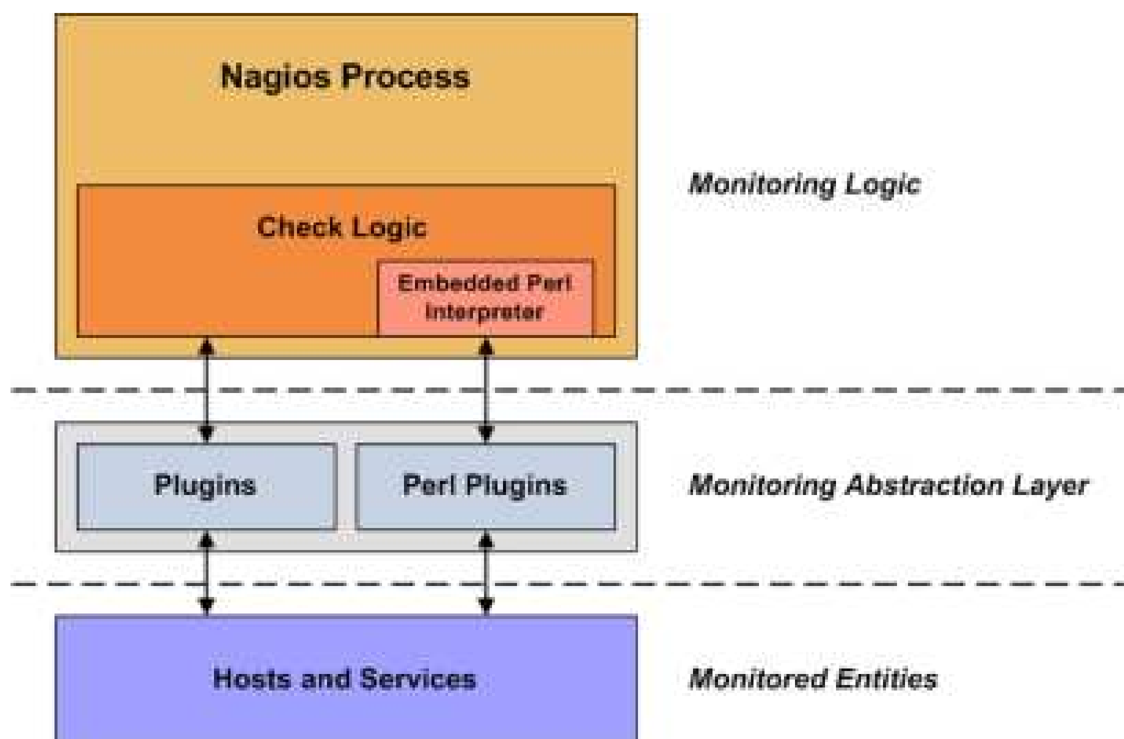


Figure 2.3: Nagios Core Plugin architecture diagram. Source: [11]

2.1.3 Strengths and Weaknesses

Below, the key strengths and weaknesses of the platform are listed.

2.1.3.1 Strengths

Nagios has several key advantages that contribute to its widespread adoption in the IT community, including:

- **Free and Open source**: One of the primary advantages of Nagios Core is its open-source nature, which allows users to access and modify the source code freely. This feature makes it particularly attractive to small companies that do not have to pay for a service and for large organizations, as they can tailor the software to meet specific monitoring needs without incurring licensing costs.

- **Active Community**: The Nagios community has grown to quite a decent number in the last decades, and even if it has seen some decline lately, it still actively supports and develops the software, providing extensive resources such as plugins, forums and a comprehensive knowledge base.

- **Robust and Wide Monitoring Capabilities**: Despite being a free tool, Nagios Core is as robust as many commercial solutions, offering comprehensive monitoring capabilities. It can collect a wide range of metrics, such as CPU usage, memory consumption, disk usage, and network latency, among others. Nagios Core is capable of monitoring virtually any network device, host, or server, making it an all-encompassing solution for various IT environments.

- **Extensibility and Flexibility**: Nagios Core's plugin architecture is another significant strength, allowing for extensive customization. The core system can monitor basic system metrics using standard plugins. However, as monitoring needs evolve, administrators can integrate third-party plugins or develop custom ones to extend Nagios Core's functionality. This flexibility enables the monitoring of nearly any conceivable parameter, provided it can be automated.

- **Integration Capabilities**: Nagios Core can integrate with a multitude of existing services and tools, enhancing its utility within diverse IT environments. For example, it can be integrated with email services for alert notifications, graphical tools for data visualization, dashboards for centralized monitoring, and log management tools for deeper analysis. This versatility ensures that Nagios Core can work seamlessly within existing infrastructure.

- **Licensed Solution**: Nagios also offers a paid out-of-the-box solution called *Nagios XI*, which meets the need of those with little expertise, allowing little to no effort setting up the monitoring system while benefiting from practically the same features as Nagios Core. This is Nagios main tool in the later years, and since the underlying architecture is very similar to Nagios Core, the later analysis will focus on it.

2.1.3.2 Weaknesses

While Nagios is a powerful monitoring tool, it does have several limitations that may pose challenges, particularly in more complex environments:

- **Scalability**: One of the major drawbacks of Nagios Core is its limited scalability. While it performs well in small to medium-sized networks, managing large-scale distributed networks can become cumbersome. As the network grows, administrators often find it necessary to decentralize Nagios Core by creating multiple instances, which complicates management and increases the overhead associated with maintaining the system.

- **User Interface**: Nagios Core includes a graphical user interface, but it is frequently criticized for being unintuitive and difficult to navigate. Configuring multiple hosts

and services through the GUI can be challenging, especially for users who are new to the platform. Nagios XI mostly solves this issue, offering a much more user-friendly interface and allowing administrators to perform all configurations through the GUI.

- **Requirement for Technical Expertise**: Nagios Core's flexibility comes with a steep learning curve. System administrators need advanced Linux knowledge to manually configure plugins, adjust file and directory permissions, and manage log files. Unlike Nagios XI, which automates many of these processes, Nagios Core requires significant manual intervention, which can be a barrier for less experienced users or smaller teams.

- **Complexity in Configuration**: The flexibility of Nagios Core, while beneficial, also results in a complex configuration process. The minimalistic design philosophy of Nagios often leads to a lack of user-friendly features, such as an interactive interface or automatic device discovery. Users must often rely on community-developed add-ons or plugins to overcome these limitations. However, some challenges, particularly related to distributed monitoring and multi-site configurations, remain unresolved.

- **Difficulty to Tailor Ad-Hoc Solutions**: The flexibility and reliability of the product to plugins becomes a double-edged sword: while it provides extensive community-built plugins, it fails to deliver an easy way to develop a new one, tailored to specific needs. This acts as a barrier for companies that cannot afford the time to develop an ad-hoc plugin or lack the expertise to do so.

In conclusion, while Nagios Core offers a simple and open-source solution for monitoring, it has some major drawbacks that partially come from its core structure with plugins. Nagios XI resolves some of these issues at the cost of being a commercial licensed software. While it is not open-source, it is often regarded as the evolution of Nagios Core, and nowdays there are plugins to mimic Nagios XI behavior, hence they are usually both just called Nagios, with one being the open-source solution and the other being the licensed one.

2.2 Zabbix

Developed a bit more lately than Nagios, Zabbix received a host of very significant awards, including "The Most Open Solution" by the Latvian Open Technologies Association [12] and the April 2019 Customers' Choice for IT Infrastructure Monitoring Tools on Gartner Peer Insights [13].
Zabbix is acknowledged for its great flexibility and adaptability to many other industries apart from IT, such as environmental monitoring, industrial automation, and even healthcare. This flexibility has made it the selection of choice of famous institutions like the European Space Agency and the University of Oslo, further underpinning its status as a solid and powerful monitoring solution still in wide use within the industry circles.
Its functionality is similar to that of Nagios, as both are out-of-the-box solutions. However, due to the resemblance between the two software and the fact that Zabbix is more widely adopted in non-IT-specific industries such as Energy, Healthcare, Banking and Finance, Retail, Government, and Education, this study will focus solely on Nagios, which is more aligned with the context of our case study.

Chapter 3

The Old Monitoring System at The Company

3.1 Overview

As The Company became more and more dominant in the market, there was an ever growing need for a better monitoring solution. This was mainly due to the rapidly expanding portfolio of apps that the company was managing, which counted at the time a few dozens of apps, several of which with millions of monthly users.
It goes without saying that numbers that high inevitably lead to some mistakes or service outages, which can cost upwards of tens of thousands of dollars per hour.
Given the company's rigorous standards, the monitoring system was required to approach the state of the art and offer a high degree of expressiveness. To meet this goal, Prometheus was introduced into the stack, as it is highly regarded as having over-the-top expressive capabilities.

3.2 Components

The system made use of several services, such as Prometheus or Grafana. The following section aims at giving a detailed overview of each of them.

3.2.1 Prometheus

Prometheus is an open source software application used for event monitoring and alerting. It was developed in 2012 by former Google employees working at SoundCloud and written in Go. Its main purpose was to meet their needs of multi-dimensionality, operational simplicity, scalability, and a powerful query language, all in a single tool.
Over the years, Prometheus has evolved into one of the most user-friendly monitoring services available. The project continues to thrive, supported by a vibrant community of developers and users.
It is now an independent open-source project, maintained separately from any corporate influence. Highlighting its importance and governance, Prometheus became the second hosted project under the Cloud Native Computing Foundation (CNCF) in 2016, following Kubernetes.

This software is used as a metric generator and its duty stops at exposing said metric. Therefore another system will be needed to handle the collection, aggregation and querying of the metrics.

3.2.1.1 Features

Prometheus offers several main features, such as:

- A sophisticated data model that organizes time series data using metric names and associated key/value pairs.
- The use of PromQL, a versatile query language designed to exploit the full potential of the data model.
- Independence from distributed storage systems, allowing each server node to operate autonomously.
- Time series data is collected using a pull mechanism over HTTP.
- The system supports time series data pushing through an intermediary gateway.
- Target discovery is handled through either service discovery mechanisms or static configurations.
- A variety of options for graphing and creating dashboards to visualize data.

3.2.1.2 Metrics

Prometheus collects and stores its metrics as time series data, meaning that each metric is recorded with a corresponding timestamp and optional key-value pairs known as labels, which are sometimes called dimensions. In simple terms, metrics are numerical measurements that track various aspects of an application's performance over time. The specific metrics to be monitored can vary depending on the application: for a web server, it might be request durations, while for a database, it could be the number of active connections or queries.
Metrics are crucial for diagnosing and understanding an application's behavior. If a web application experiences performance issues, metrics can assist in identifying the underlying cause. For example, if the application slows down as the number of requests increases, the request count metric can help identify the issue and guide actions such as scaling up the number of servers to handle the increased load.
The Prometheus client libraries offer four core metric types:

- **Counter**: A cumulative metric that represents a single monotonically increasing counter whose value can only increase or be reset to zero on restart (useful for number of requests, number of errors, etc.).
- **Gauge**: A metric that represents a single numerical value that can arbitrarily go up and down (useful for hardware statistics such as CPU utilization, number of replicas, etc.).
- **Histogram**: A histogram samples observations (usually things like request durations or response sizes) and counts them in configurable buckets. It also provides a sum of all observed values. A histogram exposes multiple time series during a scrape such as cumulative counters for the observation buckets, the total sum of all observed values

and the count of events that have been observed. It also provides easy access to quantiles from histograms that are especially useful when one wants to compute percentiles, such as when computing the 99 percentile of an API response time.

- **Summary**: Similar to a histogram, it also provides a total count of observations and a sum of all observed values. Furthermore, it also exposes streaming ϕ-quantiles ($0 \leq \phi \leq 1$) of observed events

3.2.1.3 Components

The Prometheus ecosystem comprises several components, many of which are optional:

- The core Prometheus server, responsible for scraping and storing time series data.
- Client libraries for instrumenting application code.
- A push gateway for handling short-lived jobs.
- Specialized exporters for services like HAProxy, StatsD, Graphite, and others.
- An alert manager to manage and process alerts.
- Various supporting tools.

Most Prometheus components are written in Go, making them easy to build and deploy as static binaries.

3.2.1.4 Architecture

Here is a diagram highlighting the architecture of the whole system.

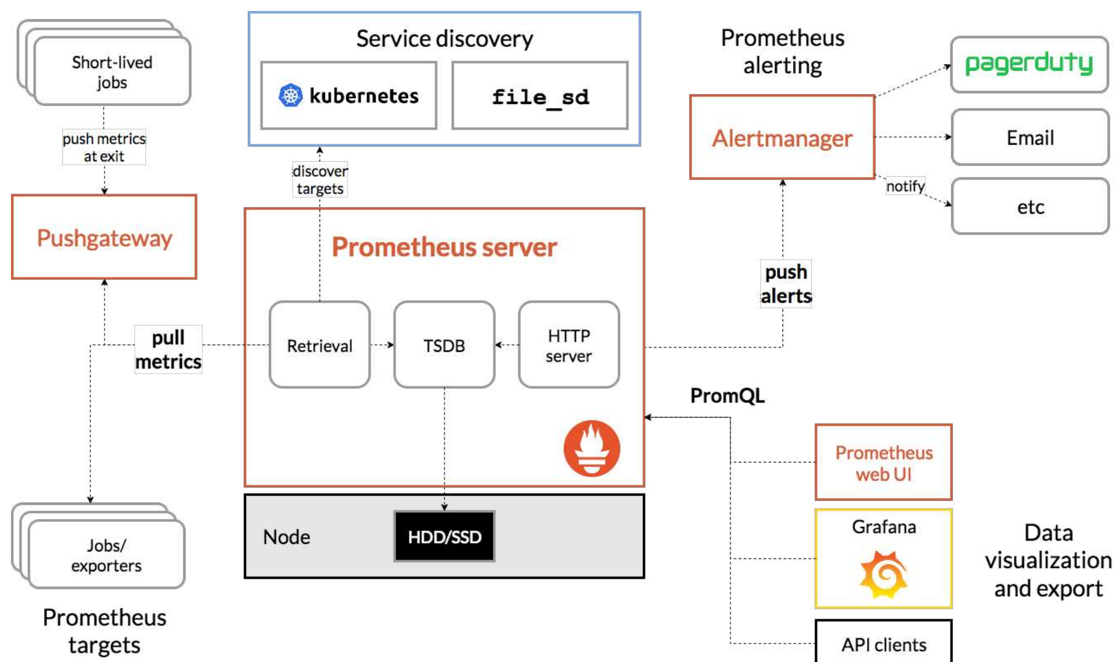


Figure 3.1: Prometheus architecture diagram. Source: [14]

In the figure we can see in orange all the main components depicted in the precious section, together with some other components that aim at demonstrating how the whole Prometheus architecture interfaces with other external tools such as Grafana for visualization of data.

The main elements are:

1. **Prometheus Server**

   - Central Component: The Prometheus server retrieves (or "scrapes") metrics data from various targets, stores this data in its time-series database (TSDB), and makes the data available through its HTTP server.
   - Data Storage: The metrics are stored on local disk (HDD/SSD) in a time-series format, allowing Prometheus to perform efficient queries on historical data.

2. **Prometheus Targets**

   - These are the sources from which Prometheus scrapes metrics, including long-running jobs and exporters that expose metrics in a format that Prometheus can consume.
   - The scraping mechanism operates on a pull model, where the Prometheus server actively retrieves data from these targets.

3. **Pushgateway**

   - Short-lived Jobs: For ephemeral jobs that may terminate before Prometheus scrapes them, the Pushgateway serves as an intermediary.
   - Push Metrics at Exit: Metrics are sent to the Pushgateway when these jobs complete, ensuring their performance data is not lost.

4. **Service Discovery**

   - Automatic Target Detection: Prometheus can automatically discover targets to scrape using service discovery mechanisms like Kubernetes or static files.
   - This helps in dynamically adjusting the monitoring scope as the infrastructure changes.

5. **Prometheus Alerting**

   - Alertmanager: Prometheus can generate alerts based on metrics data when certain conditions are met.
   - These alerts are processed by the Alertmanager, which can notify stakeholders through various channels like email or third-party services (e.g., PagerDuty).

6. **Data Visualization and Export**

   - Prometheus Web UI: Prometheus has a built-in web interface that allows users to run queries (using PromQL) and visualize metrics directly.
   - Grafana: For more advanced visualization, Prometheus data can be exported to Grafana, which supports rich dashboards and various graphing options.
   - API Clients: Prometheus also supports other API consumers, enabling integration with external systems for further data processing or visualization.

3.2.1.5 PromQL

Prometheus features a powerful query language known as PromQL (Prometheus Query Language), enabling real-time selection and aggregation of time series data. The outcomes of these queries can be visualized as graphs, displayed as tabular data in Prometheus's expression browser, or accessed by external systems through the HTTP API.
PromQL stands out for its flexibility and expressiveness, enabling users to perform complex queries that provide deep insights into the behavior of monitored systems. For instance, PromQL allows for on-the-fly calculations, filtering, and aggregation of time series data. One powerful feature is the ability to compute rolling averages over time with range vectors, which is crucial for smoothing out short-term fluctuations and observing long-term trends. For example, the query

```
avg_over_time(http_requests_total[5m] offset 5m)
```

calculates the average number of HTTP requests over a 5-minute window that occurred between 10 minutes ago and 5 minutes ago, helping to identify periods of increased traffic. The offset feature is typically very useful when creating dynamic thresholds for alerts.
Additionally, PromQL supports conditional logic, allowing users to filter and alert on specific conditions. A query like

```
rate(cpu_usage_seconds_total{job="app-server"}[1m]) > 0.8
```

could be used to monitor CPU usage, triggering an alert if any application server with the `job` attribute set to `app-server` exceeds 80% CPU utilization over the last minute. As stated before, instead of using a hard threshold of 0.8 we could instead use a dynamical one, such as "twice the amount of average CPU used in the previous day", with the query

```
rate(cpu_usage_seconds_total{job="app-server"}[1m]) >
2 * avg_over_time(rate(cpu_usage_seconds_total{job="app-server"}[1m])
[24h] offset 24h)
```

Filtering can also be done by means of regular expressions by using the operator =~, for example supposing there are three replicas named `rep-a`, `rep-b` and `replica-c`, the query

```
http_requests_total{replica!="rep-a",replica=~"rep.*"}
```

would match only

```
http_requests_total{replica="rep-b"}
```

There are many more functionalities that make PromQL a very powerful language and that are highly used, such as subqueries, functions and an extensive list of ad-hoc operators. For sake of simplicity we will avoid going in detail into these features, and instead just focus on the simpler ones. It is clear, however, that with these capabilities, PromQL enables the creation of highly specific and useful queries tailored to the needs of complex monitoring environments, which is the key of what this study aims to demonstrate.

3.2.1.6 Strengths and Weaknesses

Prometheus is well-suited for recording purely numeric time series data, making it ideal for machine-centric monitoring and highly dynamic service-oriented architectures, especially in a microservices environment where its multi-dimensional data collection and querying capabilities shine. Its design prioritizes reliability, allowing it to function effectively as a standalone system during outages without relying on network storage or remote services. However, if you need 100% accuracy for tasks like per-request billing, Prometheus may fall short, as the collected data might not be detailed or complete enough for precise measurements. In such scenarios, it's best to use another system specifically for billing data, while leveraging Prometheus for general monitoring purposes.

3.2.2 Grafana

Grafana has established as one of the main tools used for monitoring, taking the crown from previous industry standards such as Nagios. Looking at the Google Trends search for the two terms clearly reflects the change in roles the two software have had in the last decade.

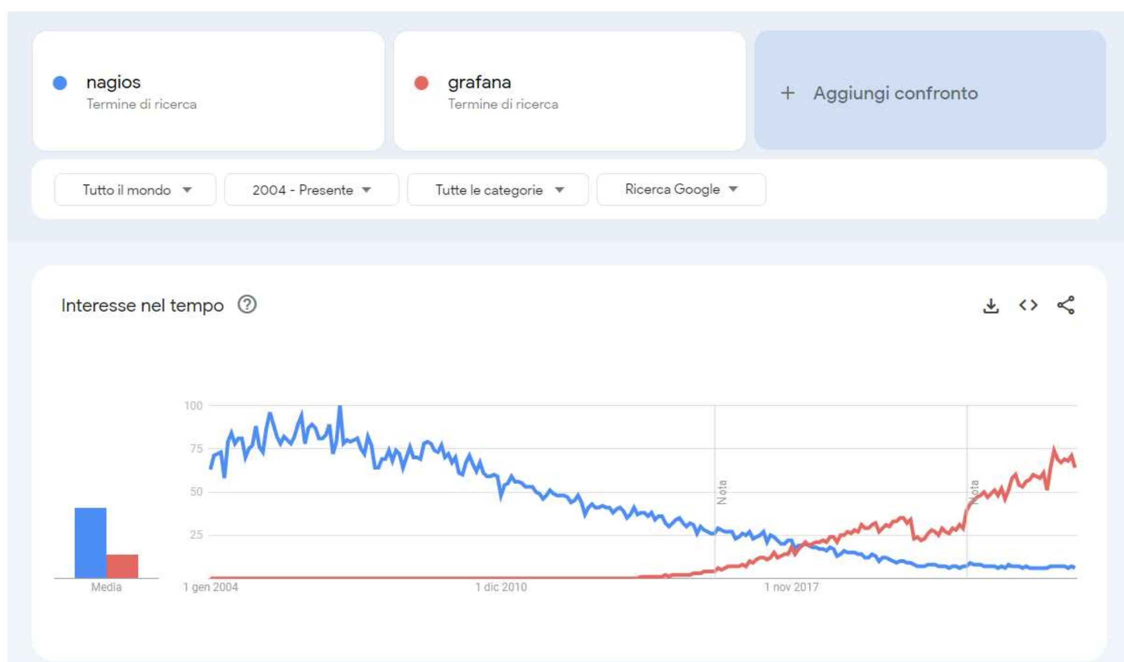


Figure 3.2: Nagios vs Grafana Google Trend

Grafana is a powerful open-source platform designed for monitoring, visualization, and alerting. It excels in providing users with the ability to create dynamic, real-time dashboards that display metrics, logs, and traces from a wide array of data sources. Grafana's flexibility in connecting to various databases and monitoring systems, including time series databases like Prometheus, makes it an indispensable tool for system administrators, developers, and DevOps teams aiming to gain deep insights into their infrastructure and application performance.

One of Grafana's standout features is its advanced visualization capabilities. Users can easily create and customize dashboards that offer a clear and comprehensive view of system metrics. Grafana supports a wide variety of graph types, such as line charts, heatmaps, and gauges, enabling users to choose the most appropriate visual representation for their data. These visualizations are not only aesthetically pleasing but also highly functional, allowing users to drill down into specific time frames, filter data by specific labels, and compare metrics from different sources on a single dashboard. This makes it easier to identify patterns, trends, and anomalies in the data, facilitating quick and informed decision-making.
The integration between Grafana and Prometheus further enhances its utility in monitoring and alerting. Prometheus, known for its robust and efficient time-series data collection, serves as a powerful backend for Grafana's visualizations. Users can seamlessly query Prometheus data within Grafana using PromQL to create detailed and insightful visualizations. This integration allows for the aggregation of metrics across multiple systems, making Grafana a central hub for monitoring complex, distributed environments.
Here is an example of what a very simple graph might look like. In this case we are interested in plotting the amount of writing to disk of an application that is monitored with Prometheus, thus the graph in Grafana will be done using PromQL.

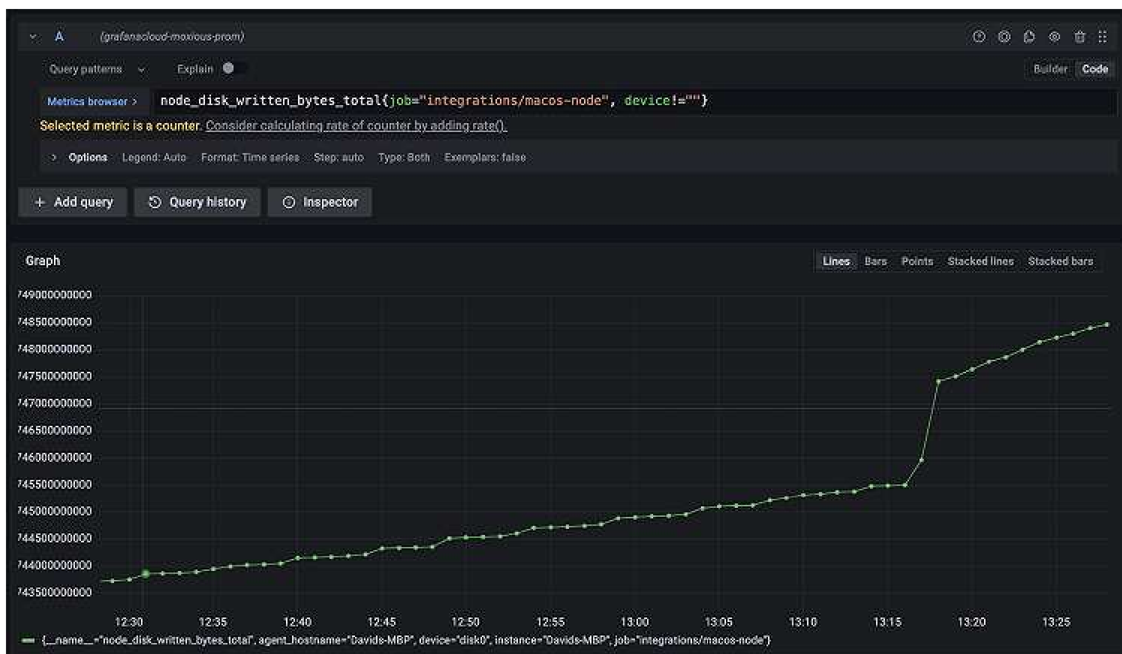


Figure 3.3: Grafana graph for number of bytes written to disk over time. Source: [15]

While this view offers some insight into system performance, it does not present a complete picture. A more accurate understanding of system performance necessitates examining the rate of change, which reveals how quickly the data being written is fluctuating. To effectively monitor disk performance, it is essential to identify spikes in activity that indicate when the system is under load and assess whether disk performance is at risk. This can be achieved by using the PromQL `rate()` function, as demonstrated below:

Figure 3.4: Grafana graph for rate of number of bytes written to disk over time. Source: [15]

As shown, the fusion of Grafana and Prometheus offers a powerful combination, capable of monitoring most systems, and using the high flexibility of the PromQL language, it can meet almost every specific need.

3.2.2.1 Alerting

In addition to visualization, Grafana's alerting capabilities are highly regarded. Grafana enables users to set up alert rules based on queries from Prometheus or other data sources. These alerts can be configured to trigger notifications via Prometheus' Alertmanager through various channels such as email, Slack, PagerDuty, or other communication tools, ensuring that teams are promptly informed of potential issues. The alerting system is highly customizable, allowing users to define thresholds, conditions, and even silence periods to prevent alert fatigue. This ensures that alerts are both actionable and relevant, helping teams to respond quickly to incidents before they escalate.

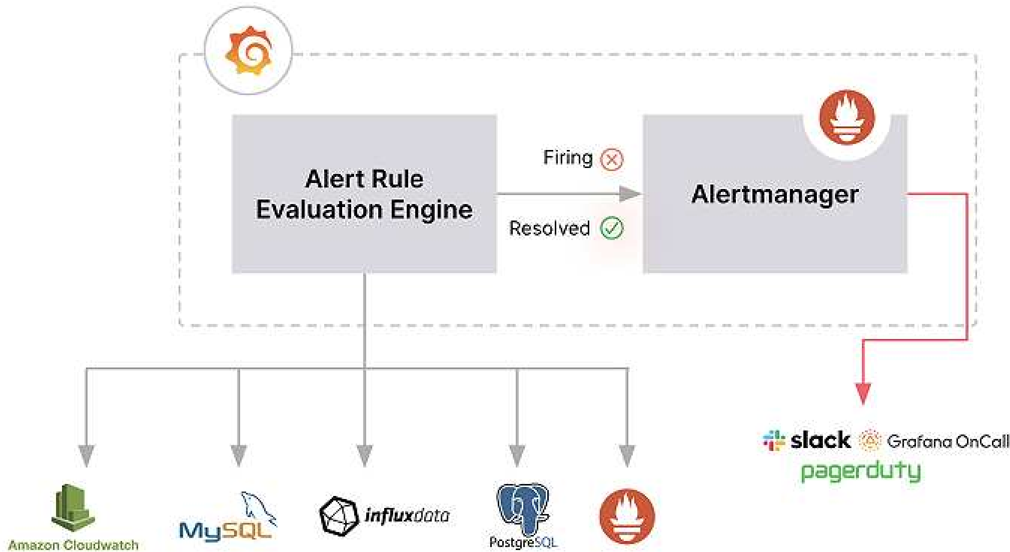


Figure 3.5: Grafana alerting default behaviour. Source: [15]

3.2.3 PostgreSQL

PostgreSQL, often referred to as Postgres, is an advanced, open-source relational database management system (RDBMS) known for its robustness, extensibility, and compliance with SQL standards. Over the years, PostgreSQL has evolved over several decades into a powerful and versatile database system used by developers and organizations worldwide.
One of PostgreSQL's key strengths is its support for a wide array of data types and complex queries, which makes it suitable for various applications, from small web apps to large-scale enterprise solutions. It supports advanced features such as transactional integrity, concurrency control, and sophisticated indexing, enabling efficient data handling and querying.
PostgreSQL's extensibility is another notable feature, allowing users to define their own data types, operators, and functions.
The system is known for its strong adherence to SQL standards while also incorporating advanced features such as JSON support for non-relational data, full-text search, and materialized views. PostgreSQL's architecture allows for high performance and scalability, making it suitable for handling large volumes of data and high-concurrency workloads.
In summary, PostgreSQL combines reliability, flexibility, and performance, making it a favored choice for developers seeking a powerful RDBMS for diverse and large scale applications, such as monitoring.

3.2.4 Cortex

Cortex is a horizontally scalable, microservices-based architecture designed to handle large-scale, distributed Prometheus metrics. The system consists of multiple microservices, each performing a specific role within the architecture. These microservices can be deployed independently and scaled horizontally to meet varying demands.
The use of a managing tool like Cortex is essential when dealing with a fleet of federated Prometheus machines, since they are not designed to handle anything after the exposure of the collected metrics

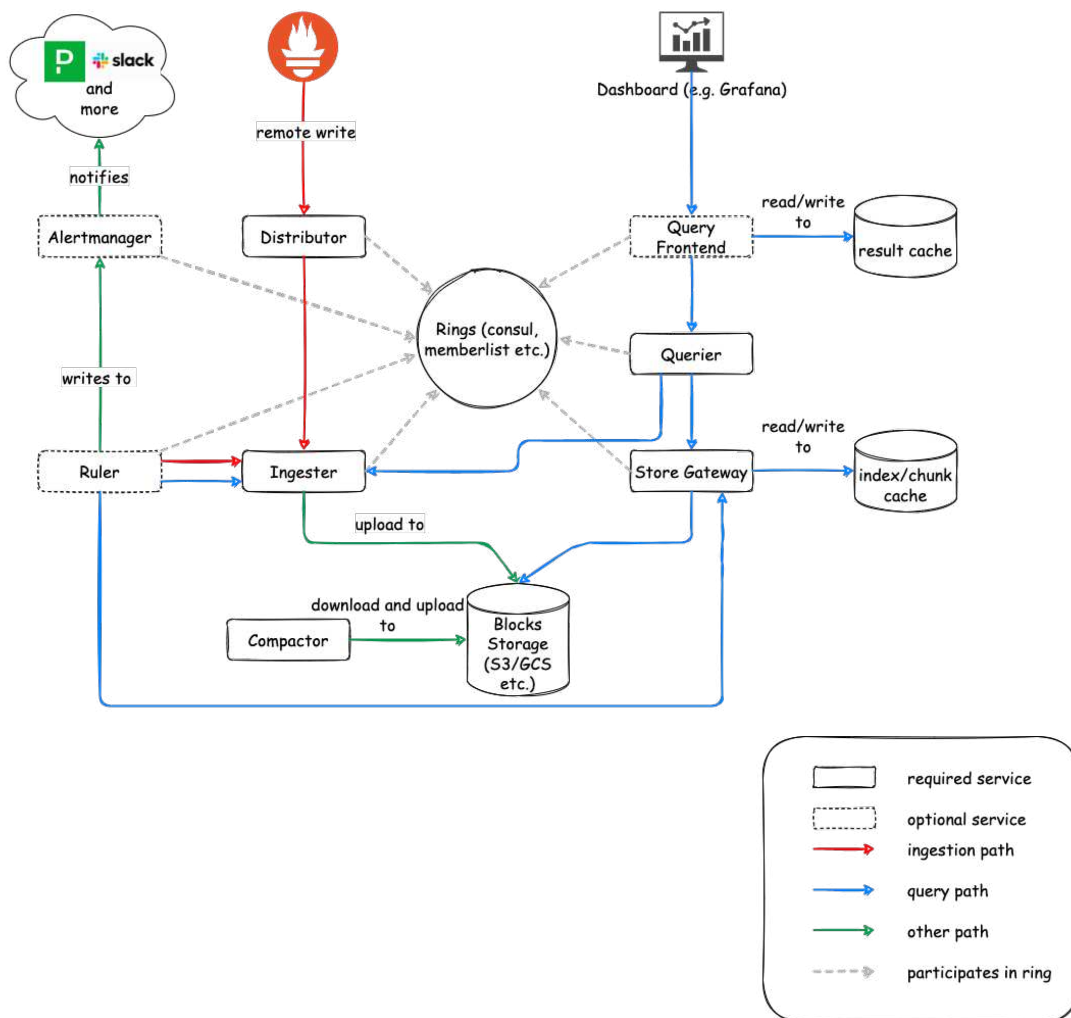


Figure 3.6: A typical Cortex deployment topology. Source: [16]

The following section aims at giving a brief overview at the main microservices used by Cortex, highlighting their different role and challenges they pose.

3.2.4.1 Distributor

The distributor service is the initial entry point for all incoming samples from Prometheus instances. It is responsible for validating and forwarding these samples to the appropriate ingesters (multiple in parallel). The distributor operates statelessly, meaning that it does not maintain any persistent data between requests. This characteristic allows it to scale horizontally with ease, ensuring that additional distributor instances can be added to handle increased traffic.
Distributors also have a feature to deduplicate incoming metrics, ensuring that even if multiple Prometheus replicas write the same metric, it is not managed twice by cortex and all the subsequent software.

3.2.4.2 Ingester

Ingesters receive and temporarily store metric samples in memory before writing them to long-term storage. This service plays a critical role in the system's write path, as it batches and compresses samples to minimize the frequency of writes to the storage backend. Ingesters are stateful due to their in-memory storage of metrics, that means horizontal scalability is very hard to achieve.
This in-memory storage introduces the risk of data loss if an ingester crashes or shuts down unexpectedly before the data is written to disk. To mitigate the risk of data loss, Cortex employs two primary strategies:

- **Replication:** Cortex typically replicates each time series across multiple ingesters. If one ingester fails, the replicated data in the other ingesters ensures that no time series samples are lost. However, in scenarios where multiple ingesters fail simultaneously, specifically those holding all replicas of a particular time series, there is a potential for data loss.

- **Write-Ahead Log (WAL):** The WAL strategy involves logging all incoming samples to a persistent disk as they are received. This ensures that even if an ingester fails, the data can be recovered by replaying the WAL during the subsequent restart of the ingester. Unlike replication, WAL provides a means of recovering all in-memory data, thereby protecting against data loss even in the event of multiple simultaneous ingester failures.

3.2.4.3 Hash ring

Distributors utilize consistent hashing combined with a configurable replication factor to determine which ingester instance(s) should receive each series, this ensures that each ingester receives a specific portion of the data.
The hash ring, which lies in a key-value store, enables consistent hashing by assigning tokens to ingesters. Each ingester is responsible for a specific range of hashes based on these tokens. When a series is received, the distributor determines the appropriate ingester by finding the smallest token value larger than the series' hash and then sends it to that instance plus the N following, where N is the replication factor. This setup ensures balanced data distribution and redundancy, mitigating data loss if an ingester fails.

3.2.4.4 Querier

Queriers handle incoming read requests, executing PromQL queries to retrieve metrics data. They fetch data from both ingesters (for recent samples) and the long-term storage. Queriers are stateless and can be scaled horizontally by adding more instances to the system. However, care must be taken to ensure that queries are balanced across available queriers to prevent performance bottlenecks.

3.2.4.5 Compactor

The compactor service optimizes long-term storage by merging multiple blocks of metrics data into larger, more efficient blocks. This process reduces storage costs and improves query performance. The compactor is stateless and can be horizontally scaled, although its role is less sensitive to high concurrency compared to other services.

3.2.4.6 Store Gateway

The store gateway facilitates efficient querying of blocks stored in long-term storage. It periodically scans the storage bucket or downloads an index to stay up-to-date with the blocks it is responsible for. This service is semi-stateful, as it needs to maintain an up-to-date view of the storage bucket. Horizontal scalability of the store gateway is supported, but care must be taken to ensure consistency across instances.

3.2.4.7 Optional Services

Cortex also includes several optional services that enhance its functionality:

- **Query Frontend:** Improves query performance by splitting and caching queries, allowing for more efficient execution across multiple queriers. It is stateless and horizontally scalable.
- **Alertmanager:** Manages alert notifications, including deduplication and routing to appropriate channels. It is semi-stateful due to its persistence of alert states and silences.
- **Ruler:** Executes PromQL queries for recording rules and alerts. It is semi-stateful, requiring careful management to avoid data gaps in the event of failures.

3.2.4.8 Challenges in Horizontal Scalability

While many of Cortex's microservices are designed to be horizontally scalable, certain challenges arise, particularly with stateful services like the ingesters and store gateways, which disallow the entire system from being totally horizontably scalable. Additionally, the use of key-value stores for coordinating state (e.g., hash rings) introduces complexity in maintaining consistent views of the system across distributed instances.
Another significant challenge is handling sudden virality spikes, which are common in apps like those operated by The Company. These spikes can force the system to either maintain a high number of replicas, resulting in underutilized resources during normal operation, or face increased latency during spikes due to insufficient scaling.

In summary, Cortex's architecture leverages microservices to achieve horizontal scalability, but careful consideration is required to manage the complexities introduced by stateful components. The system's design emphasizes redundancy and replication to mitigate the risks associated with these challenges, however, this comes at a high infrastructure cost that rises rapidly with the upscaling of the monitored system.

3.3 Architecture

The original monitoring system at The Company was built entirely in-house, without relying on any SaaS (*Software as a Service*) solution. This approach allowed the company to maintain complete independence from third-party providers.
The initial plan involved deploying a Prometheus server to scrape metrics from the VMs (*Virtual Machines*) running the services. To visualize these metrics, this Prometheus server was connected to a self-hosted Grafana instance, which enabled the building of dashboards and running of queries.
While this setup was straightforward, it had significant limitations in terms of scalability. Since the system was not stateless, it could not be horizontally scaled, and a single Prometheus server quickly became insufficient as the infrastructure grew.

The next step involved retaining Grafana for visualization, as it can be made stateless, and thus horizontally scalable, when connected to an external PostgreSQL database. All necessary security measures were configured, including Google authentication and user/team management with specific permissions.
For metric scraping, a single Prometheus server proved insufficient, necessitating multiple instances. Two approaches were considered: the first involved assigning a separate Prometheus server to each new project, managed by the respective team, with metrics forwarded to a central system. The second approach, which was ultimately chosen, involved the platform team managing Prometheus servers in a way that was transparent to the other teams. A federation of Prometheus servers was created, where each project was assigned to a federated Prometheus instance responsible for scraping metrics from its VMs via ping. However, this approach still required manual scaling, as Prometheus itself does not support horizontal scalability.

To connect Grafana with all the Prometheus instances used for scraping, a centralized point for metrics aggregation was necessary. Without this, Grafana would need to connect to each individual Prometheus instance, which is not scalable as the number of instances increases over time. This centralized point is referred to as Long-Term Storage (LTS). In the Prometheus ecosystem, two main technologies serve this purpose: Thanos and Cortex. Cortex was chosen due to its newer, state-of-the-art capabilities, and it also was fully deployed in-house.
Prometheus can interface with Cortex using a feature called "remote write", which allows Prometheus to handle metric ingestion while delegating the responsibility of metric storage and exposure to the LTS. One of Cortex microservices, specifically the ingesters, are instructed to keep on disk the data they receive for 6 hours, and then they save it to Google Drive. This value was set as a tradeoff between risk of losing data in case of failure and speed of querying. Prometheus instances send their data to Cortex through a load balancer.
In addition to the Prometheus federation, which scrapes custom metrics from individual VMs, there was also a need to ingest metrics from Google Cloud Platform (GCP) services such as Pub/Sub, Memorystore, and load balancers. This was handled by a separate Prometheus instance using Google Stackdriver exporter. This Prometheus instance also sent its metrics to Cortex, treating them as if they were their own project.
API HTTP calls were made within the same Virtual Private Cloud (VPC), rather than over the internet, as everything was hosted in-house.

Cortex operated with a load balancer that exposed two main endpoints: one for writing data (which pointed to a microservice) and one for reading data (handled by another microservice). Grafana was configured to connect to the read endpoint, while Prometheus instances were directed to the write endpoint. The remote write function in Prometheus allowed for the batching of metrics, with configurable options such as batching time, size, queue handling if Cortex was down (including retry logic and maximum retention).
Each service, including Cortex microservices, had its own replicas to ensure reliability and redundancy. While the setup was effective, there were opportunities for improvement. For instance, scaling the system significantly to handle traffic spikes could have been an option, but the associated costs were prohibitive.
Logging was managed using the native tools provided by the infrastructure provider. In the case of Google Cloud, this involved utilizing Stackdriver. This approach eliminated the need for developing a custom logging solution, saving both time and resources while still offering powerful capabilities suited to the project's requirements.

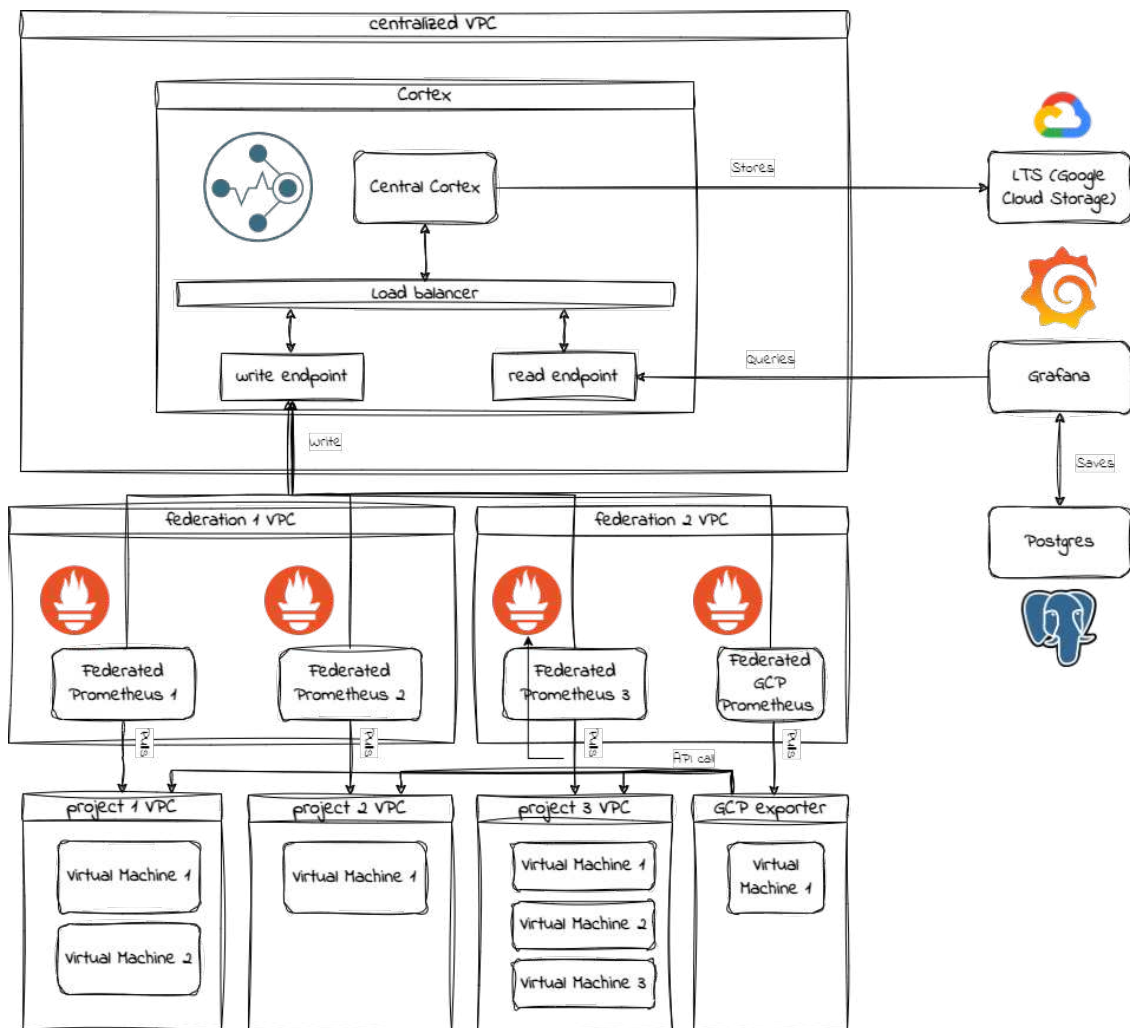


Figure 3.7: High level architecture of the old monitoring system in The Company

Each project is allocated a dedicated Virtual Private Cloud (VPC) that contains its virtual machines (VMs) and other components to be monitored. A dedicated federated Prometheus server is assigned to each project, responsible for collecting metrics from the project's com-

ponents. These Prometheus federations are housed in their own dedicated VPCs to minimize the number of network peers. This approach is necessary because Google Cloud Platform imposes a limit of 25 peers per VPC. Placing the Prometheus servers in a centralized VPC would require adding one peer for each new project. However, by assigning one peer per federation project, each federation can manage up to 24 projects.
After the Prometheus servers collect the metrics, the data is sent to the central Cortex via its write endpoint. Cortex temporarily stores this data in its ingesters before writing it to long-term storage, which, in this case, is Google Cloud Storage.
When a Grafana dashboard is accessed, it forwards the query internally to the central Cortex through its read endpoint. Cortex then retrieves the necessary data either from the ingesters or the long-term storage. Grafana uses PostgreSQL to store dashboards, configurations, and other metadata, allowing Grafana itself to remain stateless.

3.4 Limitations and Challenges

.
Several challenges arose during implementation:

- **Federation Management**: The management of federations was cumbersome. Communication between federations was done via VPC peering, which is not an elegant solution and has limitations. For instance, in Google Cloud Platform, a VPC can only be peered with a maximum of 25 other VPCs. When a federation reached its peering limit, new projects had to be deployed and assigned manually.

- **Traffic Balancing**: Balancing traffic across different federations was problematic because not all peers experienced the same traffic load. This made it difficult to manage traffic spikes effectively. Spikes were particularly challenging because Prometheus operates on a pull model (unlike newer technologies like OpenTelemetry, which use a push model). In a pull model, the Prometheus virtual machine suffers under heavy load but cannot be easily scaled up; deploying another VM would only scrape the same metric set, rather than alleviating the load on the existing one. In contrast, push systems deploy agents on every physical machine, enabling scalable metric scraping for each individual machine.

- **GCP Metrics**: The Prometheus instance responsible for reading GCP metrics, known as the GCP exporter, was shared across all The Company's projects, leading to scalability issues. Scaling this instance would result in duplicating GCP metrics. This issue was partially addressed by clustering the exporter by service (e.g., one for Redis, one for Pub/Sub), but scalability challenges persisted.

- **Cortex Scaling**: Cortex encountered difficulties with stateful microservices, particularly during traffic spikes. These spikes could cause the system to hit maximum RAM or disk capacity, making scaling a painful and manual process. Additionally, after scaling up, it was challenging to scale back down to save costs during periods of lower demand, as the handoff process was still manual.

- **Configuration Complexity**: Cortex required the management of thousands of configuration parameters, making it difficult to find and maintain an optimal setup for every situation.

- **Need for Meta-Monitoring:** A significant drawback of a self-hosted system is the need for a secondary system, known as a meta-monitoring system, to oversee the original system and ensure the reliability of the data it collects. If the meta-monitoring system is also self-hosted, a tertiary system would be required to monitor it, potentially leading to a chain of monitoring systems to achieve total reliability.

Chapter 4

The New Monitoring System at The Company

The old system had a lot of challenges and limitations, but the most restricting ones were certainly scalability and the need of constant manual labor to maintain and configure the system.
To solve these issues, a number of different solution have been proposed:

- **Use agents:** a possible improvement could have been to keep doing things in house and not use federations Prometheus but use agents (both Prometheus and Grafana) which are small, lightweight components that attach to a machine and scrape it, while using the remote-write function to still detach from the ingestion phase. With this solution it would be possible to scrape metrics of the single Virtual Machine and then, using the remote write, send them to the cortex cluster for ingestion.
  Another key improvement that agents brought to the table was the unnecessariness of deploying manually the Prometheus federations, as the agents could be integrated code-wise by each project's team, thus saving a lot of time on deploying new infrastructure for monitoring.
  While this solution would be lighter than the previous, it would still have most scalability issues, mainly those related to virality spikes.

- **Use Mimir:** another thing to consider is the possibility of using what's regarded as Cortrex successor, Grafana Mimir, which promises easier management and query performance that are up to 40 times better.

- **Use Kubernetes:** the usage of Kubernetes instead of VMs was another possible improvement, with the upside of having native support for Mimir, while with the previous solution, everything had to be done custom.

- **Decentralize GCP exporter:** GCP metrics scraping could be decentralized to a shared module that would be deployed by each team, making it highly more scalable and removing the need of a dedicated platform team to deal with the centralized architecture.

- **Pass to SaaS solutions:** Lastly, a possible solution for scaling is to pay for Software as a Service for everything that was previously done in-house. This solution will be pricier than before but will automate the process, freeing up manpower than can then be converted into more revenue.

4.1 Overview

As The Company's biggest problem is scalability, there was a big need to solve the issue using a long-term solution. After careful cost analysis, the conclusion was that the best solution among the ones listed before was to pass to SaaS solution, as well as including some of the others, such as using agents. As predictable, this includes a substantial price increase over the previous infrastructural and manpower cost, which will now be replaced by just service cost. As we will see in the analysis later, this will still end up in a net positive, given the high gain of opportunity given by freeing up engineers from this duty.

4.2 Architecture

At a high-level the system behaves like this:

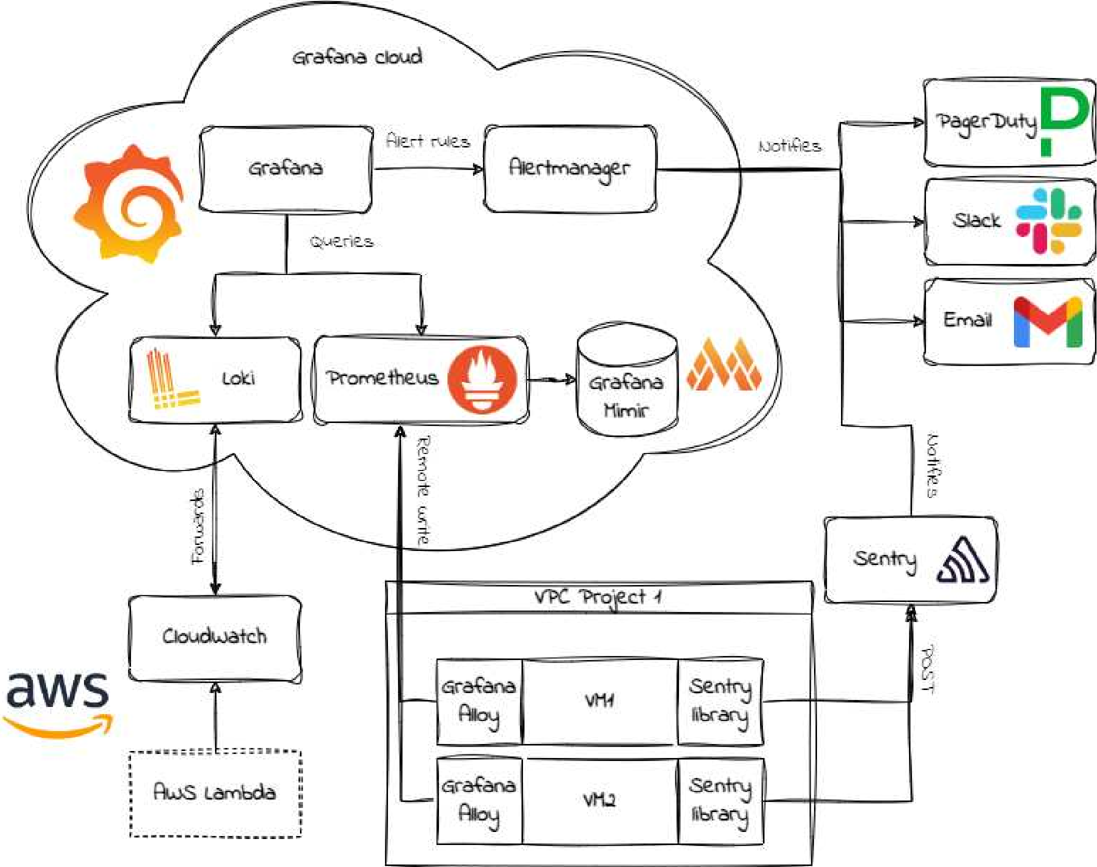


Figure 4.1: High level architecture of the new monitoring system in The Company

This solution is based on leveraging Software as a Service, shifting away from self-hosting to rely on the infrastructure provided by third-party services. Like the previous system, Grafana is still used for visualization and querying of metrics, but the difference lies in the way data is handled. Metrics are scraped and sent via Prometheus' remote-write feature to Grafana Cloud, which takes care of ingestion, storage, and authentication.

This is facilitated by the introduction of Prometheus agents, known as Grafana Alloy, which are specifically designed to handle remote writes. Another key component managed by the provider is Mimir, which now serves as the Long-Term Storage (LTS) for metrics. Mimir's multi-tenant architecture significantly mitigates issues related to virality spikes, as the provider can balance traffic across shared servers, ensuring minimal latency and downtime.
The use of Grafana Alloy agents also addresses scalability challenges, as each virtual machine runs its own agent that scrapes only its respective metrics. This setup allows for more accurate load estimation and easier planning, with metrics being sent over the internet via HTTP API calls to Mimir.
For cloud service metrics, such as those from Google Cloud Platform (GCP), a shared infrastructure is deployed via Terraform, which provisions another Grafana Alloy agent dedicated solely to scraping cloud service metrics. These metrics are also remotely written to Mimir.
The logging is also handled by the system leveraging Grafana Loki and forwarding the logs directly from the provider.
Another improvement over the last system is the introduction of Sentry as a tool to trace exceptions and errors, which vastly improves error resolution.

This new system is fully horizontally scalable, with the complexity of scaling handled by the SaaS provider, reducing the need for ongoing maintenance and freeing up internal resources.

4.3 Components

Some components are the same ones from the previous model, such as Prometheus and Grafana, with the difference than now they would be entirely managed by the provider of service, Grafana Cloud.
However, this new model also introduces a few more components, to better tackle logging, error tracing and ephemeral jobs.
In the following section the new software will be presented, with the aim of giving a brief understanding of each component, clarifying its job, as well as its strengths and weaknesses.

4.3.1 Mimir

Grafana Mimir is an open-source software project designed to provide scalable, long-term storage for Prometheus. Its core strengths include:

- **Massive scalability:** Grafana Mimir features a horizontally-scalable architecture that can be distributed across multiple machines, enabling it to process significantly more time series data than a single Prometheus instance. Internal tests indicate that Grafana Mimir can handle up to 1 billion active time series.

- **Comprehensive metric aggregation:** Grafana Mimir allows for queries that aggregate series from multiple Prometheus instances, providing a global perspective on system metrics. Its query engine extensively parallelizes query execution, ensuring even high-cardinality queries are executed rapidly.

- **High availability:** Grafana Mimir replicates incoming metrics, preventing data loss in the event of machine failures. Its horizontally scalable architecture also supports zero-

downtime restarts, upgrades, or downgrades, ensuring uninterrupted metrics ingestion and querying.

- **Native multi-tenancy:** Grafana Mimir's multi-tenant architecture allows data and queries to be isolated for different teams or business units, enabling them to share the same cluster. Advanced limits and quality-of-service controls ensure that resources are distributed fairly among tenants.

- **Cost-effectiveness and durability of metric storage:** Grafana Mimir utilizes object storage for long-term data retention, leveraging this cost-effective and highly durable technology. It is compatible with various object storage implementations, including AWS S3, Google Cloud Storage, Azure Blob Storage, OpenStack Swift, and any S3-compatible storage solution.

- **Easiness to installation and maintenance:** Grafana Mimir is accompanied by extensive documentation, tutorials, and deployment tools that make it easy to get started. Using its monolithic mode, Grafana Mimir can be deployed with a single binary and no additional dependencies. Once deployed, the system includes best-practice dashboards, alerts, and runbooks that simplify monitoring its health.

4.3.2 Loki

Grafana Loki is a horizontally scalable, highly available, and multi-tenant log aggregation system inspired by Prometheus. While Prometheus focuses on metrics, Loki specializes in logs and collects them via a push model rather than pull.
Loki is designed for cost efficiency and scalability. Unlike traditional logging systems, it does not index the contents of logs but instead indexes metadata associated with logs through a set of labels for each log stream.
A log stream consists of logs that share the same labels. These labels are crucial for efficiently locating log streams within the data store, making a well-defined set of labels essential for effective query execution.
Log data is compressed and stored in chunks within an object store, such as Amazon Simple Storage Service (S3) or Google Cloud Storage, or alternatively, on the local filesystem for development or proof-of-concept purposes. The use of a minimal index and highly compressed chunks simplifies operations and significantly reduces the cost of using Loki.

4.3.2.1 Loki Stack

A typical Loki-based logging stack comprises three key components:

- **Agent:** This component, such as Grafana Alloy or Promtail (which is distributed with Loki), is responsible for scraping logs, converting them into log streams by appending labels, and pushing these streams to Loki via an HTTP API.

- **Loki:** The central server that ingests, stores, and processes logs. Loki can be deployed in various configurations.

- **Grafana:** Utilized for visualizing and querying log data using LogQL language. Logs can also be queried through the command line using LogCLI or directly via the Loki API.

Figure 4.2: Loki logging stack. Source: [15]

Together, these components form a robust and scalable logging stack, enabling efficient log management, real-time querying, and comprehensive visualization within modern observability frameworks. This is, in fact, the logging stack used by most companies, as its simplicity makes it easy to deploy without compromising its power. Despite its ease of use, it remains a highly effective and robust tool.

4.3.2.2 LogQL

LogQL is Grafana Loki's query language, drawing inspiration from PromQL while being specifically designed for log data. Unlike PromQL, which focuses on metrics, LogQL is tailored to handle logs, allowing for efficient and powerful log aggregation and filtering. LogQL is capable of handling both simple log searches and complex data processing, making it an essential tool for anyone working with large volumes of log data.
The key features of the language are:

1. **Two different types of queries:**

   - Log queries: These queries retrieve the content of log lines. They are similar to using a distributed grep, where logs are filtered based on specified criteria. For example, you could search for all logs related to a specific service:

     ```
     {service_name="auth-service"} |= "error"
     ```

     This query retrieves all logs from the auth-service that contain the word "error".

   - Metric Queries: These queries extend log queries by enabling calculations on the query results, such as counting log occurrences or calculating the rate of log entries. For example, you could calculate the rate of error logs in the last minute:

     ```
     sum(rate({service_name="auth-service", level="error"}[1m]))
     ```

     This metric query sums the rate of logs labeled as errors for the auth-service over the past minute.

2. **Binary operators:** LogQL supports various binary operators, allowing for arithmetic, logical, and comparison operations. These operators make it possible to perform complex calculations and data manipulations directly within queries.

   - Arithmetic Operators: These include basic mathematical operations such as addition, subtraction, multiplication, and division. For instance, to double the rate of log entries, you could use:

     ```
     sum(rate({app="web"}[5m])) * 2
     ```

     This query doubles the rate of logs generated by the web app over the last 5 minutes.

   - Logical and Set Operators: Operators like and, or, and unless allow you to combine or exclude log streams based on specific criteria. For example, to find logs that are common between two applications:

     ```
     rate({app=~"frontend|backend"}[1m]) and
     rate({app="backend"}[1m])
     ```

     This query returns the intersection of logs between frontend and backend apps over the past minute.

   - Comparison Operators: Operators such as $==, !=, >, <, >=$, and $<=$ allow for filtering based on specific conditions. For example, to find logs where a specific condition is met more frequently than another:

     ```
     sum(rate({app="frontend", status="500"}[1m])) >
     sum(rate({app="frontend", status="200"}[1m]))
     ```

     This query checks if the rate of 500 status logs is higher than that of 200 status logs for the frontend app.

3. **Pattern Matching:** LogQL introduces pattern matching operators ($|>$ and $!>$) that simplify the filtering of log lines. These operators make it easier to search for specific patterns within logs without requiring complex regular expressions. For example, to find all logs that contain a specific pattern:

   ```
   {service_name="distributor"} |> "<_> level=debug <_>
   msg="POST /push.v1.PusherService/Push <_>"
   ```

   This query matches logs where the service is distributor, and the log contains a debug level entry related to a specific POST request.
   Conversely, to exclude logs matching a certain pattern:

   ```
   {service_name="distributor"} !> "<_> level=debug <_>
   msg="POST /push.v1.PusherService/Push <_>"
   ```

   This query excludes logs that contain the specified pattern.

4. **Keyword modifiers:** LogQL uses keyword modifiers like `on` and `ignoring` allow users to fine-tune the scope of operations by either focusing on specific labels or excluding them during matches. For example, when comparing log counts across machines, you might ignore the `machine` label in the matching process with a query like:

```
max by(machine) (count_over_time({app="foo"}[1m])) >
bool ignoring(machine) avg(count_over_time({app="foo"}[1m]))
```

   This query returns the machines where the total count within the last minute exceeds the average value for the `foo` app.
   Similarly, the `group_left` and `group_right` modifiers allow for many-to-one or one-to-many vector matches, adding further flexibility. For instance, to calculate the percentage of total requests for different HTTP status codes within an app, you might use:

```
sum by (app, status) (rate({job="http-server"} | json [5m])) /
on (app) group_left sum by (app) (rate({job="http-server"} |
json [5m]))
```

   This query partitions request rates by app and status, and then calculates each as a percentage of the total requests.

5. **Error handling:** Another advanced feature that might be useful is Error handling, which is designed to be as straightforward as it can be. If an error occurs during a pipeline operation, such as failing to parse a log line as JSON, the log line is not discarded but instead continues through the pipeline with an added `__error__` label. For instance, to filter out all logs that encountered JSON parsing errors, you would use:

```
{cluster="ops-tools1", container="ingress-nginx"} | json |
__error__ != "JSONParserErr"
```

   This filter ensures that only logs without JSON errors are included in the results.

Overall, LogQL combines simplicity with powerful querying capabilities, making it a versatile tool for managing and analyzing logs. Whether conducting straightforward searches or executing complex aggregations and calculations, LogQL's flexibility ensures that users can gain deep insights from large volumes of log data. This adaptability, combined with its error-handling features and advanced modifiers, solidifies LogQL as an indispensable part of any robust logging stack, empowering users to monitor, troubleshoot, and optimize their applications effectively.

4.3.3 Sentry

Sentry is a popular open-source monitoring and error-tracking tool designed to help developers identify, diagnose, and resolve issues in real-time across various environments and platforms. It's primarily used to monitor the performance and health of applications, ensuring that developers can proactively address potential problems before they impact users. The main features of Sentry include:

- **Application Monitoring:** Sentry provides comprehensive application monitoring by collecting data on performance metrics, errors, and transactions. It integrates with various languages and frameworks, allowing it to monitor applications written in JavaScript, Python, Ruby, Java, and more. This broad compatibility makes Sentry a versatile tool for developers working across different tech stacks.

- **Real-Time Alerts:** Sentry can send real-time alerts to developers when an issue is detected. These alerts can be configured based on the severity of the issue, allowing teams to prioritize responses to critical errors. Notifications can be sent via email, Slack, or other communication tools, ensuring that the right team members are informed as soon as a problem arises.

- **Contextual Information:** Sentry captures detailed context about each error or performance issue, including the stack trace, environment, user actions, and the specific lines of code that triggered the issue. This contextual information helps developers quickly pinpoint the root cause of a problem, reducing the time needed to diagnose and fix it.

- **Release Tracking:** Sentry allows developers to track errors and performance metrics by release version. This feature helps teams correlate issues with specific deployments, making it easier to identify and address problems introduced in recent updates. Release tracking also supports regression detection, alerting developers if a previously resolved issue reappears in a new version.

- **Performance Monitoring:** Beyond just error tracking, Sentry offers performance monitoring features that allow teams to track the performance of their applications. This includes measuring response times, throughput, and other key performance indicators (KPIs) to ensure the application is running smoothly and efficiently.

- **User Feedback:** Sentry can capture user feedback directly within the application, allowing users to report issues they encounter. This feedback is linked to the corresponding error or performance issue in Sentry, providing additional context from the user's perspective.

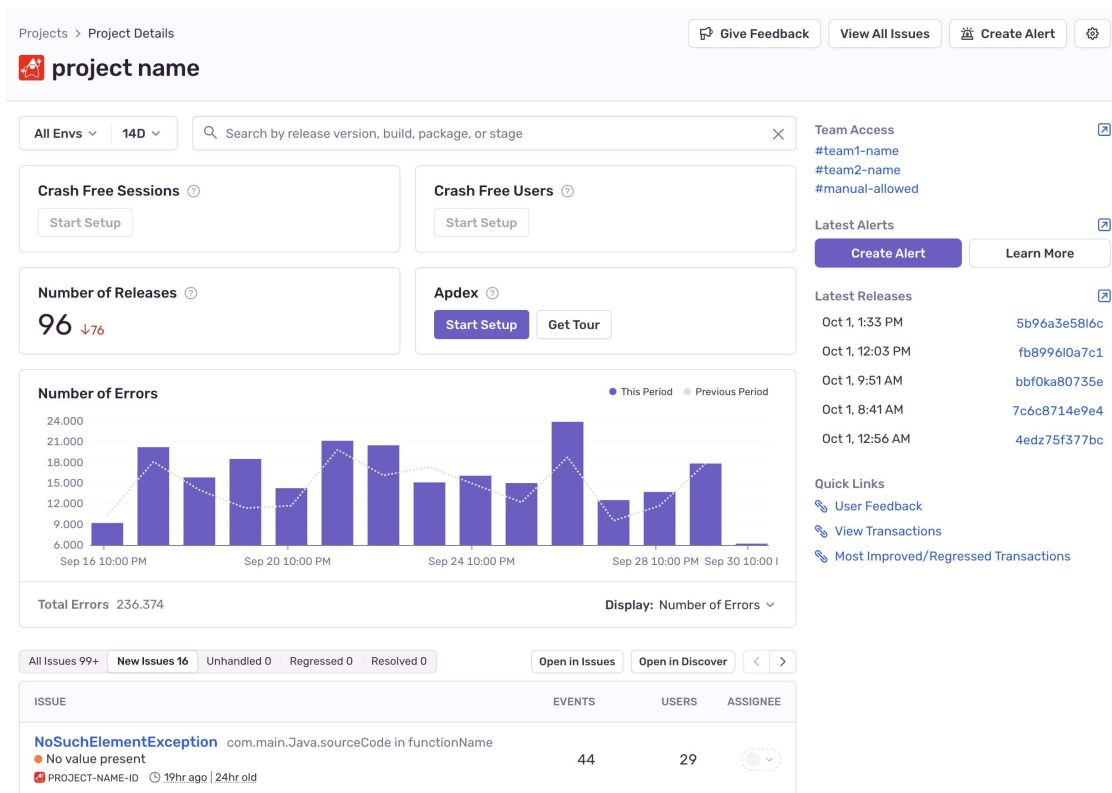


Figure 4.3: Sentry main project page

4.3.3.1 Error Monitoring

One of Sentry most loved and useful features regards error tracing and monitoring. Unlike traditional logging, which provides only with a trail of events, some of which are errors, but many are simply informational, Sentry is fundamentally different because it is focused on exceptions, capturing only application crashes.
Sentry's error tracing is designed to automatically do this job of detecting and tracking errors and exceptions in applications. When an error occurs, Sentry captures detailed information, including the error message, stack trace, environment details, and user actions leading up to the error. It groups similar errors into issues, helping developers prioritize and focus on critical problems by giving a scope of how frequent each on them is.
Sentry also provides tools for debugging, such as source maps and traceability to the specific lines of code causing the issue.
Here are some key features that it offers in this field:

- **Detailed Error Tracking:** Sentry automatically tracks errors and exceptions in your application as they occur. It captures detailed information about each error, including the error message, stack trace, affected users, and the environment (e.g., browser, OS, version). This automated tracking helps developers identify issues that might go unnoticed during manual testing.

- **Issue Grouping:** Sentry groups similar errors together into issues, helping to reduce noise and avoid alert fatigue. By grouping errors that share the same root cause, Sentry ensures that developers can focus on resolving the underlying problem rather than getting bogged down by multiple reports of the same issue.

- **Error Prioritization:** Sentry provides tools to prioritize errors based on their impact. For instance, errors affecting many users or causing critical failures are highlighted, allowing teams to address the most pressing issues first. Sentry's dashboards and reports provide insights into the frequency and severity of errors, aiding in effective prioritization.

- **Exact line of code Traceability:** Sentry's detailed error reports include stack traces that point to the exact lines of code where the error occurred. This traceability is invaluable for debugging, as it allows developers to quickly identify and fix the specific part of the codebase responsible for the issue. Additionally, Sentry can capture local variables, breadcrumbs (user actions leading up to the error), and the application state at the time of the error, providing a comprehensive view of the error's context.

- **Source Maps and Symbols:** For applications written in languages that compile or minify code (like JavaScript or C++), Sentry supports source maps and symbolication. This feature translates the minified or compiled code back into its original source form in the error reports, making it easier to understand and debug the issue. This is essential for languages such as TypeScript, which gets compiled to JavaScript. Without this feature the stacktrace and code snippets would be shown in the compiled version, which is much less readable.

- **Error Resolution Workflow:** Sentry supports a robust workflow for resolving errors, including assigning issues to team members, adding comments, and marking issues as resolved or ignored. This workflow integration helps teams manage the error resolution process efficiently, ensuring that all errors are tracked and addressed systematically.

- **Integration with DevOps Tools:** Sentry integrates with various DevOps tools like GitHub, Jira, and other issue-tracking systems, enabling seamless transitions from error detection to resolution. When an error is detected, it can automatically create a ticket in your issue tracker, linking the error report directly to the task in your development workflow.

Overall, Sentry is a comprehensive monitoring tool that excels in error tracking and performance monitoring, offering developers the insights they need to maintain application health and improve user experience. Its powerful error monitoring capabilities, combined with its integrations and detailed context capture, make it an essential tool for modern development teams who want to efficiently deal with runtime issues.

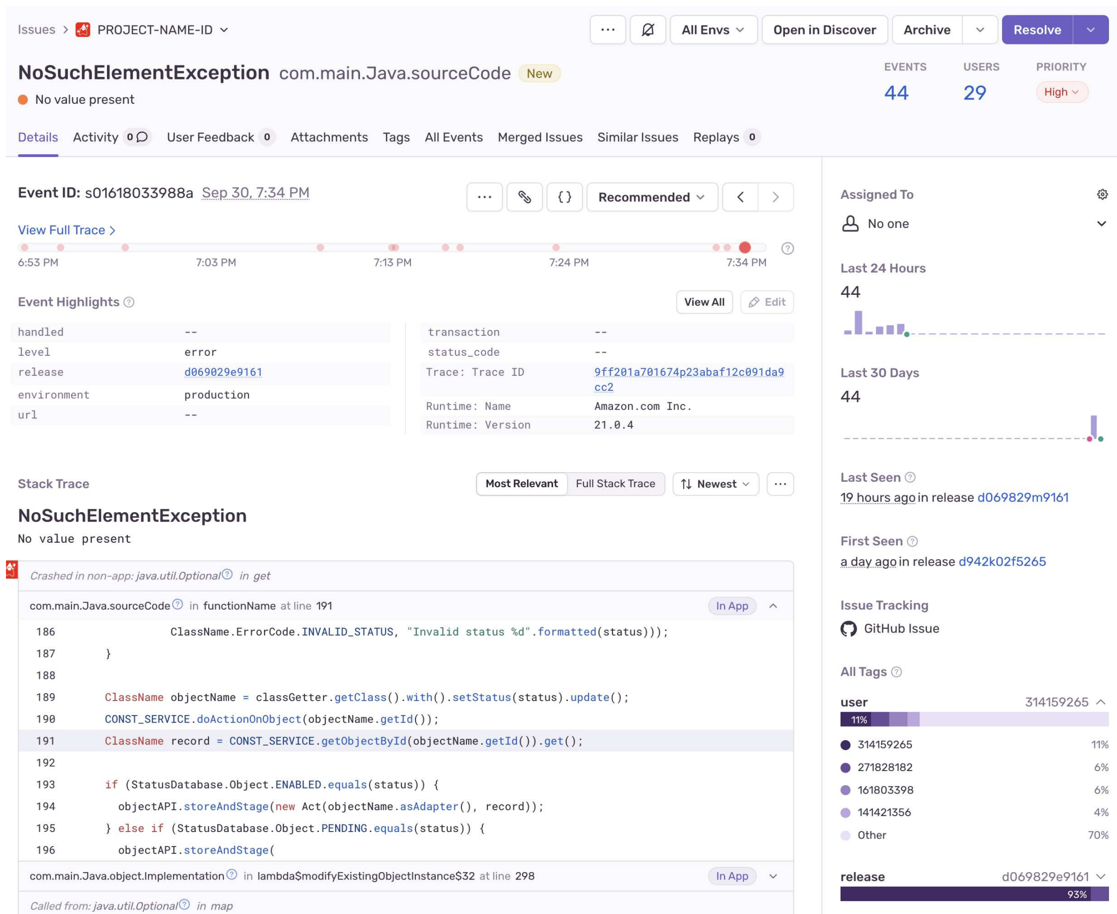


Figure 4.4: Sentry page for specific error

4.3.4 PagerDuty

PagerDuty is a leading incident management platform designed to help organizations enhance their operational reliability and respond effectively to system outages, critical issues, and performance degradations. With its focus on real-time operations and incident management, PagerDuty enables teams to detect, prioritize, and resolve incidents before they escalate, minimizing downtime and improving customer experiences. It also has a built-in feature that allows for oncall rotations, scheduling team members to be responsible to acknowledge and act upon incidents occurring during their schedule. Below is a detailed overview of PagerDuty as a service, with an emphasis on its ease of use, effectiveness, and its status as a de facto standard in the incident management space.

4.3.4.1 Key Features and Strengths

1. **Ease of Use:** PagerDuty is known for its intuitive and user-friendly interface, which enables teams to quickly adopt and configure the platform with minimal effort. Its design emphasizes simplicity, ensuring that even teams with limited experience in incident management can set up monitoring and escalation policies efficiently.

    - **Quick Setup and Integration:** PagerDuty provides out-of-the-box integrations with over 600 tools, making it easy to integrate with services that teams already use, such as Prometheus, Alertmanager, AWS CloudWatch, Datadog, and others. This pre-built integration ecosystem allows users to seamlessly incorporate PagerDuty into their existing workflows without complex configurations.
    - **Web and Mobile Access:** The platform is accessible via both web and mobile applications, enabling users to manage alerts, incidents, and on-call rotations from anywhere. This mobility ensures that team members can respond to incidents in real time, even when they are not at their workstations.
    - **On-Call Scheduling and Escalation Policies:** PagerDuty's on-call management capabilities are straightforward to configure. Teams can define on-call schedules and escalation paths that automatically notify the right people at the right time. The platform's drag-and-drop interface allows for easy modification of schedules and rules without the need for technical expertise.

2. **Effectiveness in Incident Management:** PagerDuty excels in streamlining the incident response lifecycle—from detection to resolution. The platform offers a variety of features that enable teams to respond quickly and efficiently to issues, reducing mean time to recovery (MTTR).

    - **Real-Time Alerts and Notifications:** PagerDuty ensures that critical alerts are delivered in real time across multiple communication channels, including SMS, email, push notifications, and voice calls. This multi-channel approach ensures that incidents do not go unnoticed, and the appropriate team members are alerted promptly, minimizing delays in response.
    - **Advanced Incident Routing:** One of PagerDuty's core strengths is its advanced incident routing and automated escalation policies. Incidents are routed to the right on-call team based on the severity and type of issue. If an incident is not acknowledged within a specified time frame, it escalates automatically to the next level of response, ensuring that issues are addressed without unnecessary delays.
    - **Collaboration and Automation:** PagerDuty integrates seamlessly with collaboration tools like Slack and Microsoft Teams, allowing teams to coordinate responses to incidents directly within their chat applications. Additionally, PagerDuty supports automation workflows, enabling teams to define automated responses to incidents, such as restarting services, running diagnostics, or triggering predefined remediation scripts.
    - **Post-Incident Analysis and Reporting:** After an incident is resolved, PagerDuty provides comprehensive post-mortem analysis and reporting. These reports include details on incident timelines, response times, and any delays in acknowledgment or resolution. The insights gathered from these reports help teams identify areas for improvement in their incident response processes.

3. **De Facto Standard and Broad Integrations:** PagerDuty has established itself as the industry standard for incident management, particularly in DevOps and IT operations environments. Its widespread adoption across organizations of all sizes is a testament to its reliability and effectiveness. One of the primary reasons for its popularity is the extensive range of integrations it offers.

   - **Integration with Prometheus' Alertmanager:** PagerDuty is tightly integrated with Prometheus, a leading monitoring and alerting toolkit, and Alertmanager, which handles alerts generated by Prometheus. This integration allows organizations to easily send Prometheus alerts to PagerDuty, where they can be escalated and managed using PagerDuty's powerful incident response features. The integration ensures that alerts generated from Prometheus, based on specific monitoring conditions, are immediately routed to the appropriate on-call engineers, enabling prompt action.
   - **Cloud, Monitoring, and IT Service Management (ITSM) Tools:** PagerDuty integrates with a broad range of cloud providers (AWS, Azure, GCP), monitoring tools (Datadog, New Relic, Zabbix and Nagios), and ITSM platforms (ServiceNow, Jira, Zendesk). These integrations allow organizations to centralize their monitoring, alerting, and incident management processes within a single platform, reducing complexity and enhancing operational visibility.
   - **Customizable APIs and Webhooks:** For teams with specific requirements, PagerDuty provides customizable APIs and webhooks that allow developers to create custom workflows and integrations. This flexibility ensures that PagerDuty can be adapted to fit virtually any organization's needs, regardless of how unique their environment may be.

4. **Reliability and Scalability:** PagerDuty is built to scale with organizations of any size, from small startups to large enterprises. Its cloud-based architecture ensures high availability and reliability, critical for incident management platforms. As organizations grow and their systems become more complex, PagerDuty can handle the increased volume of alerts and incidents without degradation in performance.

   - **Global Reach and Distributed Teams:** PagerDuty supports organizations with distributed teams across different geographies. It allows for configuring region-specific alerting rules and schedules, ensuring that the right teams in the right time zones are notified of incidents.
   - **Robust Analytics and Insights:** PagerDuty's analytics capabilities provide deep insights into operational performance. Teams can track key metrics such as incident frequency, response times, and service reliability, helping them identify trends and continuously improve their processes.

Overall, PagerDuty has established itself as a market leader in incident management due to its ease of use, powerful incident routing, and broad integration capabilities. Its seamless integration with popular tools like Prometheus' Alertmanager, combined with its effectiveness in managing real-time incidents, makes it the de facto standard in the industry. Whether for a small team or a global enterprise, PagerDuty ensures that critical issues are addressed efficiently, reducing downtime and improving overall operational reliability.

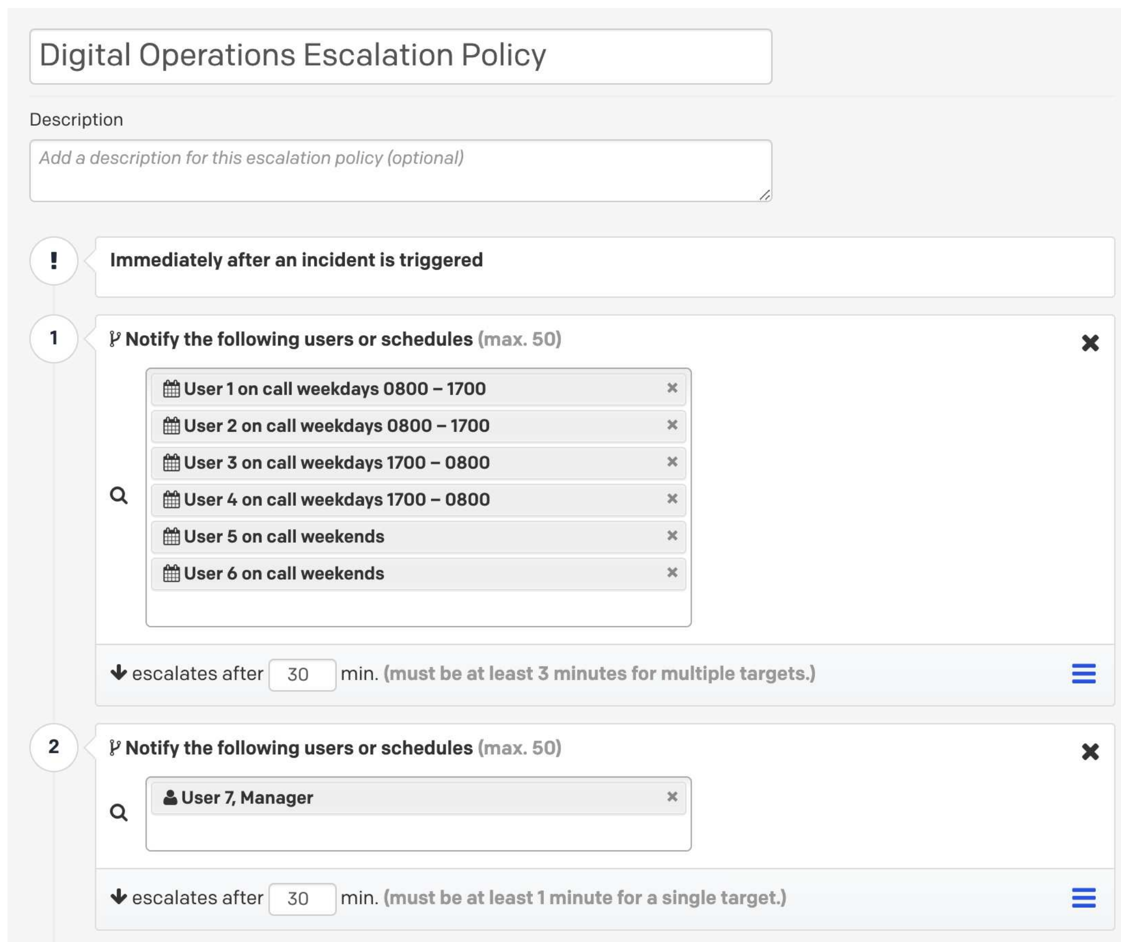


Figure 4.5: PagerDuty incident escalation policy example. Source: [17]

4.4 Logging

Previously, logs were managed using the integrated tools provided by the cloud platform. With the transition to Grafana Cloud, it is now advantageous to use Grafana Loki due to its high performance, customizability, and expressiveness.
Although it is possible to create log streams directly in Loki through specific agents that must be deployed on each virtual machine, it was decided to continue using the native log streams from the cloud provider's tools. These logs are then forwarded to Grafana Cloud. This approach is more cost-effective, as the native logging services, such as AWS CloudWatch or GCP Stackdriver, are often offered at minimal cost, and are sometimes included in the service packages provided to companies.
The forwarding is handled via a sink, specifically AWS Kinesis Firehose, which provides an endpoint where logs from AWS or GCP are sent. In GCP, however, logs are read by Grafana Alloy agents configured to scrape logs instead of metrics, using a technology called Promtail (provided by Grafana Cloud) to tail logs and forward them to Loki. Once the logs are sent to Loki, they are deleted from the native logging tool to avoid unnecessary storage costs. This is done using an exclusion rule for GCP and by setting a low retention time in AWS.
The primary advantage of this solution is that it represents a seamless upgrade from the previous system and requires minimal effort to implement. Only the log forwarding process needs to be configured, with Kinesis Firehose and Promtail being relatively simple to set up.

4.5 Challenges

This system faces several challenges, particularly due to the pull-based nature of Prometheus. A key issue arises with ephemeral jobs, such as AWS Lambdas or Google Cloud Functions. These are short-lived functions that execute a task and terminate quickly, often running on-demand. This creates a significant problem for a pull-based system like Prometheus, which relies on persistent instances to scrape for metrics. Ephemeral jobs do not exist long enough for Prometheus to collect metrics, nor do they have a stable identifier for the scraper to track.

4.6 Ephemeral Jobs

To address the challenges presented by ephemeral jobs, there are three main approaches:

- **Switch to Push Systems:** In recent years, push-based systems have become more advanced and are now widely adopted across the industry. OpenTelemetry, which has emerged as the new industry standard, supports push mechanisms. With this, agents can be attached to ephemeral jobs, allowing metrics to be pushed at the end of each job's execution. Grafana Cloud already supports OpenTelemetry as part of its ecosystem, making integration straightforward. However, adopting this approach would require a company-wide shift to OpenTelemetry, as maintaining both push and pull systems simultaneously would be inefficient. This significant infrastructure overhaul should be evaluated based on the extent to which ephemeral jobs contribute to overall system performance.

- **Use Pushgateway:** Prometheus offers a specific solution for ephemeral jobs through its Pushgateway component, which acts as middleware between ephemeral jobs and the Prometheus scraper. This involves integrating the Pushgateway into the job's code, allowing metrics to be explicitly written and then exposed for scraping via an endpoint of the Pushgateway itself. While relatively easy to implement, Pushgateway adds execution time to ephemeral jobs, given by the POST call made to push the metrics. Since these jobs are billed based on duration, this additional time can significantly increase costs, especially for jobs that typically run within a few hundred milliseconds. In such cases, doubling execution time may lead to unacceptable cost increases.

- **Use Log-Based Metrics:** A more lightweight solution is leveraging log-based metrics, particularly if the system is already configured to handle logs. Ephemeral jobs naturally emit status logs upon completion to native logging solutions, such as AWS CloudWatch. By adding a few extra logs, the jobs can emit metrics without the overhead of a POST call, as required by Pushgateway. These logs are then forwarded to Grafana Loki, where they can be queried using LogQL. This solution seamlessly integrates with existing infrastructure, especially if Loki is already in use for log management. However, forwarding logs to Loki consumes a log subscription filter, which may be a concern in environments with strict limits (e.g., AWS allows only two filters per log group, which can quickly become exhausted if one filter is used for Loki forwarding and another for Prometheus exporting).

The system presented in this chapter ultimately opts for the third solution of log-based metrics since the additional costs associated with Pushgateway are prohibitive, and switching to OpenTelemetry would require substantial time and effort. Log-based metrics offer an

efficient, low-cost alternative that integrates smoothly with the existing use of Loki for log management. The use of a log subscription filter is acceptable within the current infrastructure constraints.

4.7 Alerting

Grafana offers native support for alerting, a key feature to have in any monitoring system to be able to promptly respond and evaluate incidents.
How the system works is easily defined in these three steps that run periodically:

1. Grafana Alerting queries data sources and assesses the conditions specified in the alerting rules specified by the user.

2. If the condition evaluates as true, then an alert is fired with that condition.

3. Firing alerts are sent through the designated notification channels while they remain in the "firing" state. If the alerting rule later evaluates to false, an additional notification is triggered, indicating that the issue has been resolved, either automatically or through human intervention.

Figure 4.6: Grafana Alerting diagram. Source: [15]

This can then be linked with Prometheus' Alertmanager as shown in figure 3.5

4.7.1 Alerting rules

An alert rule is the key component of an alert. It consists of one or more conditional queries, so that the metric that has to be monitored is constantly evaluated against a static or dynamic threshold, after which the alert is fired.
For example this alert rule

```
avg_over_time(http_requests_total{level="error"}[5m]) > 100
```

will fire if there are more than 100 error http calls in the last 5 minutes.
An alert rule can also be configured to fire if the specified query evaluates to true for a specific amount of time, for example if we set an alert rule like this

```
avg_over_time(http_request_time) > 1000
```

will fire anytime we have a HTTP request that takes over than 1 second to finish, and it is undesirable to have an alert raised every time this happens. We can then set a condition to raise the alert only if this keeps happening for 5 minutes, indicating a service outage.

4.7.1.1 Alert rule evaluation

There are two criteria determining the evaluation of each rule: evaluation group and pending period.
While the previous alert rule might be useful to notice any network issues, there are always a small portion of calls that will fail due to various reasons, because no service is 100% reliable. So we might want to specify that the latter query will trigger an alert rule only if the condition is persistent for 5 minutes, this is what's called a pending period. It is then clear how useful this feature really is to avoid spamming the alerting channels with single alerts instances.
Each alert rule is associated with an evaluation group. You can either assign the alert rule to an existing group or create a new one.
Each evaluation group has a specified evaluation interval, which dictates how often the alert rule is assessed. For example, evaluations can be scheduled every 10 seconds, 30 seconds, 1 minute, 10 minutes, and so on.

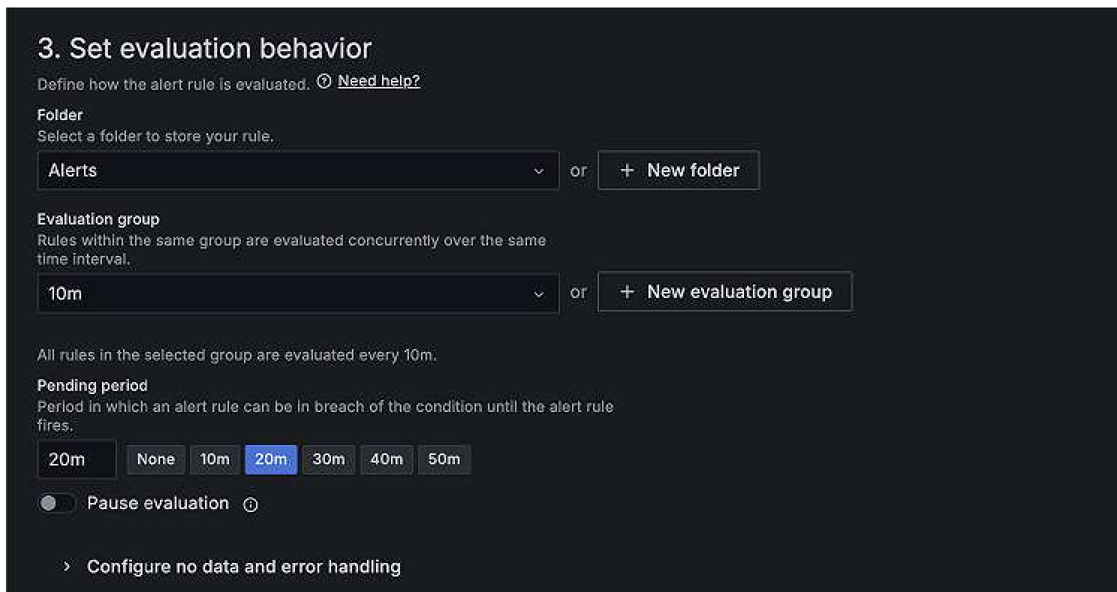


Figure 4.7: Grafana Alerting rule configuration. Source: [15]

Here is a simple diagram explicating the steps an alert rule can take during its evaluation cycle.

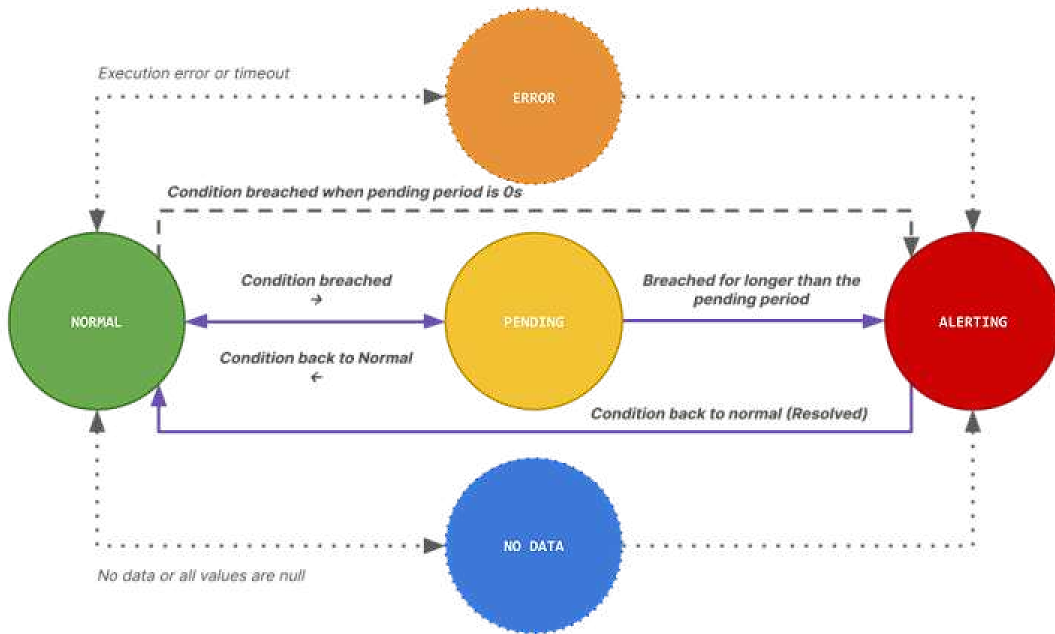


Figure 4.8: Grafana Alerting rule cycle diagram. Source: [15]

In this context:

- **Normal**: The state where the alerting condition is not met.
- **Pending**: The state where the alerting condition is met, but the pending period has not yet elapsed.
- **Alerting**: The state where the alerting condition has been met and the pending period has passed.

The transitions between these states are as follows:

- **Normal to Pending**: This transition occurs when the alerting condition is met, but the pending period has not yet elapsed. If the alerting condition resolves during the pending period, the state will revert to normal.
- **Normal to Alerting**: An alert can transition directly from normal to alerting if the pending period is set to 0, if there are errors in the query, or if there is no data. Conversely, an alert will move from alerting to normal if the alerting condition is resolved.
- **Pending to Alerting**: This transition occurs when the pending period expires and the alerting condition is still met. Transitioning from alerting back to pending is not allowed.

4.7.1.2 Alerting instances

Each rule can produce multiple instances of a specified alert, this is because each one is tied to the corresponding time series. This is especially useful when monitoring multiple replicas of the same server. One single copy of an alert rule can be enough to fire for each individual server.

	State	Labels	Created
>	Normal	alertname=CPU Usage cpu=0 type=cpu	2022-04-27 15:32:55
>	Normal	alertname=CPU Usage cpu=1 type=cpu	2022-04-27 15:34:35
>	Normal	alertname=CPU Usage cpu=2 type=cpu	2022-04-27 15:31:35
>	Normal	alertname=CPU Usage cpu=3 type=cpu	2022-04-27 15:34:55

Figure 4.9: Grafana Alerting instances. Source: [15]

4.7.1.3 Notifications

In the alert rule it is also possible to specify the channels for the notification (called contact points), making it possible to divide alerts by priority, i.e. setting the more serious alerts to fire more direct streams such as phone calls, SMS or third party services such as PagerDuty, and the less important ones via slack or email.
The message that gets sent is also completely customizable by using templates, and it is also possible to populate it with variables, for example the first alerting rule of the section can be customized to send the message

```
There have been ${avg_over_time(http_requests_total{level="error"}[5m])}
error requests in the last 5 minutes!
```

giving the receiver a sense of how much the threshold has been surpassed, which is again very useful to captivate the magnitude of the alert.

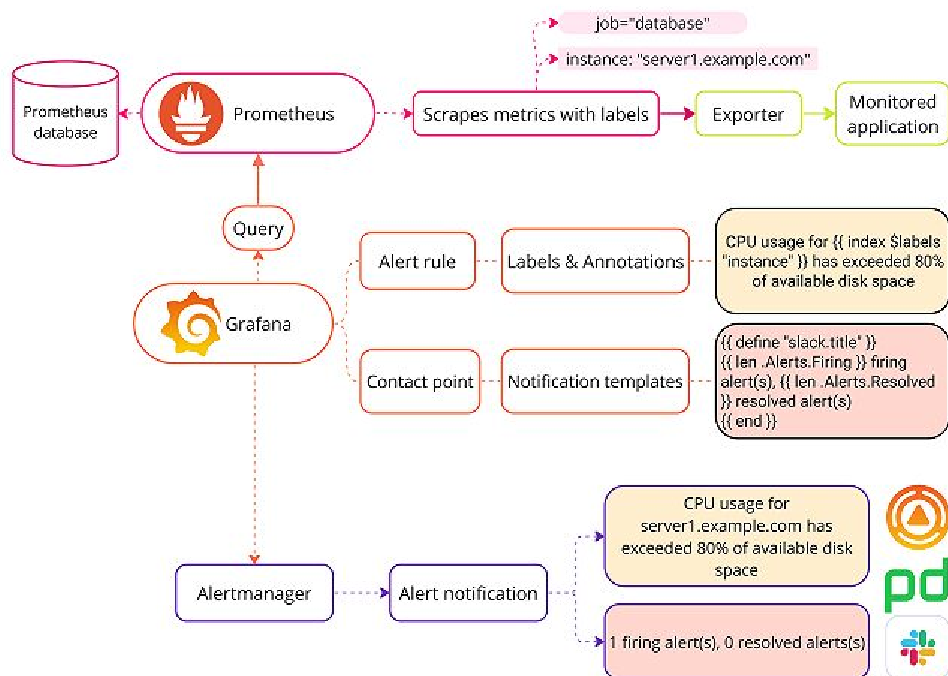


Figure 4.10: Grafana Alerting templates diagram. Source: [15]

In the figure above:

- **Monitored Application**: This refers to any web server, database, or service that generates performance metrics. For instance, a web server may generate metrics regarding request rates, response times, and other performance indicators.
- **Prometheus**: Prometheus is responsible for collecting metrics from the monitored application. For example, it may scrape metrics from a web server, with each metric associated with specific labels such as `instance` (representing the server's hostname) and `job` (denoting the name of the service being monitored).
- **Grafana**: Grafana serves as a querying interface to retrieve metrics data from Prometheus. For example, one may define an alert rule to track request rates of the web server over time, utilizing labels like `instance` to create templates or annotations within the alerting system.
- **Alertmanager**: As part of the Prometheus ecosystem, Alertmanager is responsible for managing alert notifications. For example, if a particular web server exceeds a predefined request rate threshold, Alertmanager can dispatch an alert notification to platforms such as Slack or via email. The `instance` label is interpolated to include the actual server name within the notification.
- **Alert Notification**: Upon fulfillment of an alert rule's conditions, Alertmanager issues notifications to various channels, including Slack or PagerDuty. These notifications may contain specific details derived from the labels associated with the alerting rule. For instance, an alert triggered by high CPU usage on a specific server may include the server name (as identified by the `instance` label), the percentage of disk usage, and the threshold that was breached.

All these components come together to produce a notification with interpolated values, for example from the generic template

```
CPU usage for {{ index $labels "instance }} has exceeded 80% of available
disk space
```

becomes

```
CPU usage for server1.example.com has exceeded 80% of available
disk space
```

This example query uses the built-in set of variable `$labels`, which contains automatically all the labels from the query in the alert rule. Another default variable is `$value` which is a table containing the values of the alerting rule, making possible to also return the actual value of the resulting query in the following way

```
CPU usage for {{ index $labels "instance }} has exceeded 80% of available
disk space: {{ index $values "A" }}
```

which becomes

```
CPU usage for server1.example.com has exceeded 80% of available
disk space:  81.32%
```

Note that the table of `$values` is indexed by ID, which by default are letters in alphabetical order, that's why in the last query we need to use `$values "A"`.
Finally, it is also possible to instruct the Grafana to batch issues together to reduce the spam of notifications. This is useful in big systems, where each single component is monitored, so the developers can know which component is affected and which is not within a single notification.

4.7.1.4 Recording Rules

A recording rule enables the pre-computation of frequently used or resource-intensive expressions, storing the results as a new time series. This is particularly useful for running alerts on aggregated data or optimizing dashboards that repeatedly query complex expressions.
By querying the newly created time series, performance improves significantly, especially for dashboards that refresh frequently.

4.7.1.5 Architecture

Grafana Alerting is based on the Prometheus model for designing alerting systems, which consists of two primary components:

- **An alert generator** that evaluates alert rules and sends both firing and resolved alerts to the alert receiver.
- **An alert receiver**, also called Alertmanager, which handles incoming alerts and manages their notifications.

While Grafana Alerting does not rely on Prometheus as its default alert generator, since it supports multiple data sources beyond Prometheus, it can still utilize Prometheus for alert generation, as well as external Alertmanagers, depending on the configuration (see fig 3.5).

4.8 Error Rate monitoring

When dealing with error rates, SLOs (Service Level Objectives) are used because they provide a more reliable and context-sensitive approach to monitoring service performance, as demonstrated in [18].
In order to understand SLOs, the concept of SLIs (Service Level Indicators) must also be introduced.

4.8.1 SLIs

A Service Level Indicator is a key performance metric, such as availability, used to assess the health or performance of a service over time. SLIs quantify actual performance results, often expressed as a fraction or percentage, such as 99.9% system availability (or 0.999).
In Grafana SLO the creation of precise SLIs is facilitated by its ratio query builder, while also supporting custom PromQL queries, allowing for flexibility in defining performance metrics tailored to specific service needs.

4.8.2 SLOs

A SLO defines the target that an SLI must achieve to ensure an acceptable level of service. This target specifies the threshold of performance that is acceptable for users, such as the percentage of error-free responses within a given time frame that prevents noticeable service degradation. For example, an SLO might dictate that 99.9% of requests to a web service must be processed without server errors over a 28-day period.

When defining SLOs, it is important to acknowledge that aiming for 100% availability is neither practical nor cost-effective. As availability approaches 100%, the complexity and associated costs increase disproportionately. Therefore, it is essential to account for a margin of error, commonly referred to as an error budget.

To illustrate, consider a credit card processing application. Although the company has committed to 99.95% availability in its contracts, this figure may not fully capture customer expectations. Using the SLO framework, the company can specify its availability goals more precisely. In this case, the SLO could state that 99.97% of credit card validation requests must return without a 500 error and within 100 milliseconds. Since validation needs to be instantaneous, allowing e-commerce platforms to inform customers of errors before completing a purchase, this SLO ensures that the service meets user needs. The corresponding SLI would be the percentage of credit card validation requests that succeed within the specified time frame and without errors.

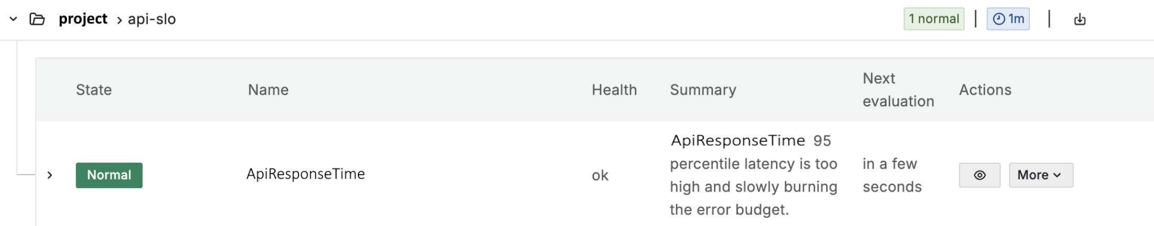


Figure 4.11: Grafana SLO alerting rule

Unlike pure error rate tracking, which triggers alerts immediately when any error occurs, potentially causing false alarms due to momentary spikes (e.g., a one-second spike in errors), SLOs define acceptable error thresholds over a longer period, effectively doing a "weighted" or time-based evaluation of the error rate. This approach acts like an integral over time, considering the cumulative impact of errors rather than reacting to transient issues, which ensures that alerts are raised only when the service's reliability genuinely deviates from the defined objective. This reduces alert fatigue and focuses attention on incidents that truly affect user experience.

SLOs are essential not only for operational alignment but also for ensuring transparency with users. They allow service providers to set realistic performance targets in collaboration with users through SLIs and SLAs (Service Level Agreements), making it clear that no system can be 100% reliable. By defining an acceptable level of error, users have a clear understanding of the expected service quality and the degree of occasional failure that is considered normal. This transparency builds trust, as users know upfront the reliability they can expect and what constitutes acceptable degradation, helping to manage their expectations and avoid unrealistic assumptions about flawless service delivery. This transparency is particularly important when dealing with SaaS, as any enterprise can know in advance the error rate to expect from that service and act accordingly, for example by integrating a retry policy in case of error.

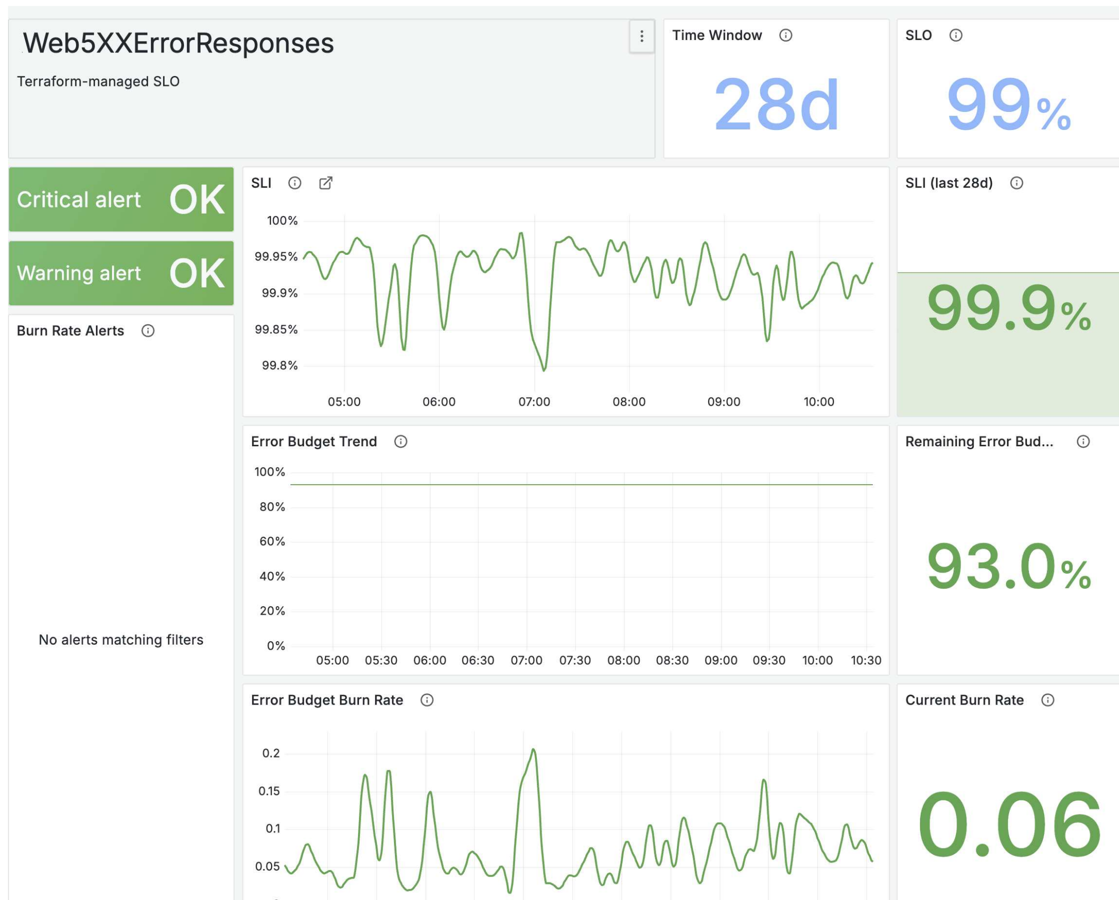


Figure 4.12: Grafana SLO dashboard

When implementing a SLO strategy, it is important to familiarize with several key concepts:

- **Error Budget:** An error budget represents the allowable margin for failure when measuring service performance. It reflects the difference between actual and desired performance levels. For instance, using the previous example, the error budget would be the gap between perfect service (100%) and the service level objective (SLO) of 99.97%. This budget can be expressed as a percentage (e.g., 0.03% allowable failures) or as a specific amount of time (e.g., 12 minutes of non-compliance over 28 days).

- **Burn Rate:** The burn rate is the rate at which a service consumes its error budget, essentially indicating how quickly the service is approaching its allowed margin of failure. When setting an SLO, such as 99.5% availability, the remaining 0.5% accounts for acceptable errors or slower responses. A burn rate of 1 implies that the service is exhausting its error budget at the exact rate permitted by the SLO, while a slower burn rate (e.g., 0.75) suggests that the service is outperforming the SLO. Conversely, a faster burn rate indicates that the service quality is below expectations, and corrective actions may be required.

- **Alerting on Burn Rate:** SLO-based alert rules trigger notifications when the service is at risk of depleting the error budget within the defined timeframe. This approach ensures that alerts are generated only when business objectives are at risk, avoiding unnecessary notifications for every minor threshold breach. Grafana, for example, provides both fast and slow burn rate alerts. Fast burn rate alerts are designed for critical issues like outages, which may quickly consume the error budget and require

immediate attention. In contrast, slow burn rate alerts are triggered by less-urgent issues, such as minor bugs or network slowdowns, and may prompt the creation of a ticket in systems like Jira, ServiceNow, or GitHub for further investigation.
For instance, if the service is consuming 2% of the error budget per hour, Grafana will trigger an urgent alert, notifying an on-call engineer through tools such as PagerDuty to address critical events like system outages or hardware failures. However, if the service is burning through 0.8% of the error budget per hour, a less-critical alert may be issued, prompting the team to open a ticket for non-urgent issues, such as a performance degradation caused by a bug.

- **Fast-Burn Alert Rule:** Over short time intervals, Grafana triggers alerts when the burn rate is particularly high, signaling severe conditions such as outages or hardware failures that require immediate response.
- **Slow-Burn Alert Rule:** Over longer time intervals, Grafana sends alerts when the burn rate is moderate, indicating ongoing issues that, while less critical, still require timely resolution.

A good SLI for Error rate is simple and easy to read, for example

```
requests_total{code!~"5.."} / requests_total
```

for availability, and

```
requests_duration_seconds_bucket{code!~"5..", le="1.0"} /
requests_duration_seconds_count{code!~"5.."}
```

for latency.

Chapter 5

Analysis

The following chapter aims at giving some theoretical analysis of the three systems presented. The goal is to identify when each system is better to use then the others, so that it becomes easier to choose one of the alternatives, depending of each specific need and condition.
Given the very high amount of variables that such a situation may comprise, in this study we will simplify by fixing some of them. This simplification should not impact the quality of the results, since the chosen variables to be fixed, should only impact the scale of the results, and not their respective rankings.

Throughout the analysis, the Nagios system will be referred as *System 1* or just (1), The in-house solution as *System 2* or just (2) and the Grafana Cloud solution as *System 3* or just (3).

5.1 Theoretical Analysis

The first section of the analysis will just be theoretical. The aim of this section is to give results based on common knowledge or publicly available information.
In the following section, those results will be tested against practical scenarios, to see if the results match or if in reality some systems perform better or worse then expected.

5.1.1 Evaluation Criteria

The systems will be evaluated based on a set of simple, objective metrics that can be clearly measured across different monitoring solutions. The following criteria are used to guide the analysis:

- **Cost:** This is the primary criterion for comparison, as financial considerations are critical for any organization. Cost encompasses both the initial setup and ongoing maintenance. Different systems have distinct cost structures: for instance, an in-house system incurs significant setup costs, both in terms of infrastructure and human resources, while a SaaS solution, like Grafana Cloud, typically requires minimal upfront investment.

- **Scalability:** Scalability is crucial for companies expecting growth in the coming years. A system must be capable of handling increasing demands without requiring substantial changes. Systems that lack inherent scalability can hinder future expansion, necessitating costly overhauls.

- **Resilience to Traffic Spikes:** This criterion assesses the system's ability to handle sudden and extreme increases in traffic, which are common for businesses susceptible to viral trends or unpredictable demand surges. While related to scalability, this focuses specifically on the system's ability to manage short-term, high-intensity traffic loads, which can easily overwhelm less resilient architectures. Not being able to capitalize on these spikes represents a significant loss of potential revenue for companies, and must therefore be prevented.

- **Ease of Maintenance:** Particularly relevant for smaller companies with limited technical staff, this criterion considers how simple a system is to maintain. Systems that require minimal intervention and expertise are preferable for organizations without dedicated IT personnel. For example, a small company might favor a solution that operates with minimal knowledge, configuration and maintenance effort.

Typically, monitoring systems are assessed based on their Mean Time To Response (MTTR). However, for the three systems analyzed here, the MTTR is not measured. This decision is based on the fact that MTTR is highly dependent on specific implementations, and all three systems employ an alert manager to dispatch notifications about potential issues. Consequently, the variability in MTTR would largely be influenced by the nature of the specific issues encountered. Additionally, setting up all three systems to undergo identical scenarios for measurement purposes presents significant challenges and might be infeasible to do with low degrees of uncertainty.

5.1.2 Parameters

As mentioned at the beginning of this chapter, evaluating monitoring systems in detail requires accounting for a wide range of variables. Even just estimating costs involves data such as the volume of metrics ingested per day, the number of time series processed, the frequency of queries (both for visualization and alerting), the number of machines being monitored, the number of active users, and the amount of egress traffic. While allowing these variables to fluctuate would offer a more flexible analysis, enabling companies to tailor decisions based on their specific circumstances, for simplicity, most of these factors will be fixed at realistic values representative of high-scale operations.

5.1.2.1 Fixed parameters

The fixed parameters will be:

- volume of metrics ingested: this parameter will be set at 100TB/month.

- number of time series active: this parameter will be set at 20M/month.

- volume of logs ingested: this parameter will be set at 30TB/month.

- number of machines: this parameter will be set at 10000.

5.1.2.2 Variables

The variables of the model will be:

- System used: This parameter reflects the choice of either of the three systems presented between (1), (2) and (3).

- Cost of opportunity: This parameter reflects the additional revenue a company can generate if the engineers, who would otherwise be dedicated to maintaining the in-house monitoring system, are reassigned to work on other projects. By redirecting these resources, the company can capitalize on new opportunities or enhance existing initiatives, thus increasing overall revenue potential.

5.1.3 Cost analysis

To analyze cost we need to define a function that captures the overall economical net of a company. This will simply be:

$$Cost_{tot} = Cost_{system} + Cost_{opp}$$

Where the total cost of the system is computed as the cost of building and maintaining the system plus the Cost of opportunity of relocating resources. This way we will only capture the end result of cost, taking into account every scenario.
We now have to compute the cost of each system. To do this, we use publicly available usage-based prices, as neither GrafanaCloud or Nagios offer publicly prices for enterprises. All costs computed will be per months.

- **Cost of (1):** Nagios does not provide public, usage-based pricing. For this analysis, the most appropriate pricing estimate has been used, and prices for larger packages have been extrapolated from smaller ones.
  The out-of-the-box solution requires purchasing a Nagios XI license. For small to medium-sized enterprises, the cost for unlimited nodes is approximately \$26,000. However, given our scale, the "sitewide" package, designed for distributed and large-scale deployments, would be necessary. While no official pricing exists for this option, estimates based on online data suggest that the 10-node package costs around \$55,000. Scaling this to meet the needs of a system with approximately 10,000 machines, we estimate a cost of around \$5.5 million. This approximation follows typical pricing models and, for simplicity, the assumption of linearity of the increase in price, though the actual cost may vary significantly. It is important to note that this cost is a one-time payment for the license, and not a monthly subscription.
  As for setup, we can estimate the cost of implementation at roughly \$17,000, based on one full-time engineer working on the system for two months, assuming an average annual salary of \$100,000. In total, the cost of Nagios XI is projected to be around \$5.5 million for a one-time license plus setup costs.

- **Cost of (2):** The cost of the second system depends significantly on the specific implementation details, but a reasonable estimate can be made based on common usage patterns for this type of application and then using public prices for GCP.

  - To store 130TB of metrics and logs, using Google Cloud Storage would cost approximately \$2,600/month.
  - For Cortex servers, approximately 30 high-memory machines are required for ingesters. Based on the cost of n2-highmem-64 machines from GCP, this would total \$92,000/month.
  - Prometheus federation would require around 50 general-purpose e2-standard-8 machines, leading to a total of approximately \$10,000/month.

- The cost of maintaining and building the system would include two full-time engineers, which adds about $16,800/month, assuming an annual salary of $100,000.

Summing these components, the total cost of System 2 is approximately $121,400/month.

- **Cost of (3):** The cost of system 3 can be computed using Grafana Cloud usage-based pricing and GCP for the egress cost as follows:

  - $160,000/month for 20M active series.
  - $65,000/month for 130TB of ingested metrics
  - $11,700/month for egress traffic
  - No cost for setting up and maintaining, since it is offered as SaaS.

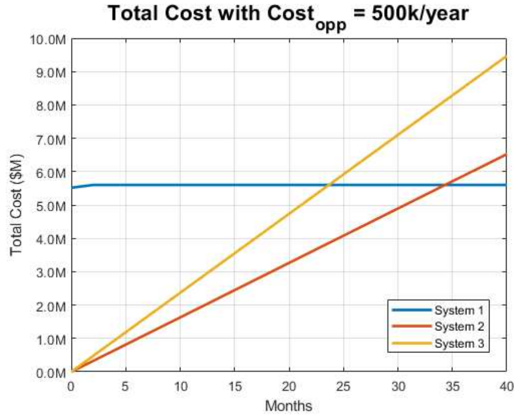
  Overall system 3 would cost $236,700/month

For simplicity, here is reported a table with the costs of the systems:

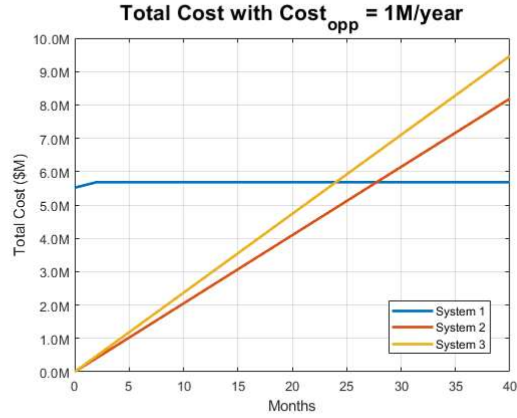
System	Components cost	Total cost
System 1	License: $5,500,000 (one-time) Setup: $17,000 (one-time)	**$5,517,000** (one-time)
System 2	Storage: $2,600/month Cortex Servers: $92,000/month Prometheus Federation: $10,000/month Maintenance: $16,800/month	**$121,400/month**
System 3	Active Series: $160,000/month Ingested Metrics: $65,000/month Egress traffic: $11,700/month	**$236,700/month**

Table 5.1: Table of total costs of the three systems

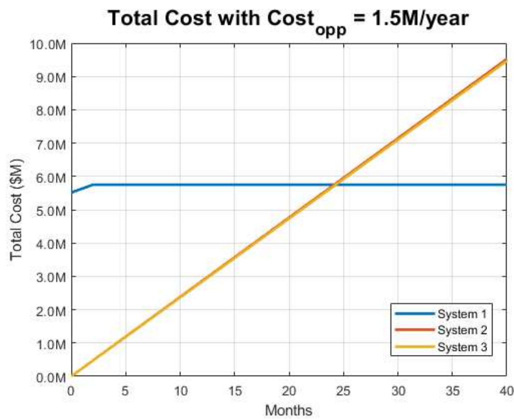
Here are the costs of the three systems plotted for four different values of Cost_{opp}: $500,000/year, $1,000,000/year, $1,500,000/year, and $2,000,000/year. Each graph represents the total cost over a period of 40 months, according to the formula defined at the beginning of this section.



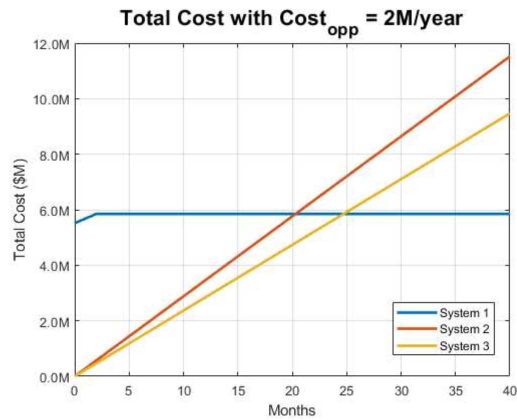
(a) Costs with Cost_{opp} = \$500,000/year



(b) Costs with Cost_{opp} = \$1,000,000/year



(c) Costs with Cost_{opp} = \$1,500,000/year



(d) Costs with Cost_{opp} = \$2,000,000/year

In the four scenarios depicted above, the analysis reveals the following insights:

- System 1 proves to be more cost-effective in the long term. This is because it involves a one-time license fee rather than a recurring monthly cost. However, it is important to note that this comes with limitations in scalability and adaptability of the monitored system. Therefore, System 1 remains the preferred choice for small to medium-sized companies with limited growth expectations or those that anticipate changing their monitoring stack infrequently.

- Both Systems 2 and 3 surpass System 1 in cost after 2 to 3 years. Hence, if the monitoring requirement is anticipated to last less than this time frame, choosing either System 2 or System 3 would be more economical.

- The cost of opportunity value at which Systems 2 and 3 are roughly equivalent is slightly less than \$1,500,000/year (∼\$1,400,000/year). For values below this threshold, System 2 is more cost-effective, while values above this threshold make System 3 the cheaper option.

A purely numerical analysis is insufficient, as companies may prioritize flexibility, scalability, and other factors beyond cost, and often will be prepared to pay a bit more for a better tool or for one that will withstand the test of time better. Therefore, additional metrics are also analyzed below.

5.1.4 Scalability analysis

Scalability is a challenging aspect to quantify objectively, so we will outline the necessary steps for scaling each system and identify any hard limits they may encounter.

- **System 1:** Scalability is generally not a concern for this system, as it is managed internally. However, performance issues arise when scaling beyond a certain number of hosts. Nagios XI is designed to handle a limited number of hosts effectively; for instance, the most expensive package supports up to 1,000 hosts. While it may perform adequately for a few thousand hosts (e.g., up to 3,000), monitoring tens of thousands of hosts, as in our case, would likely exceed its practical capabilities. Thus, for very large deployments, Nagios XI may not be the optimal choice.
- **System 2:** Scalability presents significant challenges with this system due to its limited horizontal scalability. The system requires constant manual deployment of new infrastructure, which increases complexity. Although it could theoretically be scaled infinitely with sufficient resources and manpower, cloud providers like GCP impose hard limits on various aspects, such as the number of peering connections and machines per VPC. For example, GCP has a limit of 24 peering connections per VPC, necessitating a hierarchical approach to scale beyond this. As a result, adding more than 24 squared projects requires additional layers of peering, complicating the architecture further.
- **System 3:** Scalability is managed entirely by the external provider, so it is theoretically infinite, provided the provider can accommodate the increased demand. As long as the provider supports and can handle higher traffic volumes, scalability is not a concern for this system.

5.1.5 Traffic Spikes analysis

For traffic spike analysis, it is necessary to assume a magnitude. For simplicity, assume that the spike is five times the normal traffic, thereby generating five times the load on the current machines and requiring five times the number of machines to maintain the same load on each.

- **System 1:** Traffic spikes are generally manageable if Nagios is configured properly. However, if spikes cause the number of hosts to exceed the software's limitations as discussed in the previous section, the entire system may experience performance degradation or, in the worst case, complete shutdown.
- **System 2:** Traffic spikes present a significant challenge for this system, as scalability must be handled manually. Unless the spike can be anticipated several days in advance to allow for the deployment of necessary infrastructure, there is a high likelihood that the system will collapse under increased traffic. This scenario is particularly problematic because not only would there be a substantial increase in costs to deploy five times the number of machines, but the system might fail due to its data ingestion bottleneck before any monitored systems encounter issues, resulting in a loss of visibility on potential problems.
  One potential solution is to design the system to handle up to x times the normal traffic, although this approach would significantly increase costs during normal operations.

- **System 3:** Traffic spikes are not an issue for this system, provided that the provider can handle them. The primary concern would be the associated increase in service costs. This is particularly relevant if the contract is not based on usage but is limited to a certain amount of traffic, in which case overage charges may apply.

In conclusion, the most effective solution for managing traffic spikes is System 3. It is therefore preferable if the expected traffic in the applications is not consistently constant and instead follows trends or viral waves.

5.1.6 Maintenance analysis

When maintaining a monitoring system, several factors may be of interest, such as the level of expertise required and the time commitment involved.

- **System 1:** This system is designed to function with minimal maintenance effort, making it an ideal solution for small to medium-sized businesses that may have less specialized staff or cannot afford to dedicate a full-time engineer. However, if customization to specific needs becomes necessary, it can be challenging and requires substantial IT knowledge.

- **System 2:** This system demands considerable expertise and time for both setup and maintenance. Nevertheless, it offers the greatest degree of flexibility in implementation. It is recommended for organizations that require tailored applications and have the capability to manage the associated complexity.

- **System 3:** This system requires less expertise than System 2 but more than System 1, as some components, such as agents on monitored machines, need to be deployed manually. This solution can also be used with minimal maintenance effort, as it is offered as a SaaS product and any issues can be addressed through the provider's customer support.

In summary, both Systems 1 and 3 offer a balanced compromise between required knowledge and time commitment. However, System 2 may be preferable if the organization is focused on developing specific, tailored applications, due to its higher degree of customizability.

5.1.7 Summary

For convenience, the results theorized above are reported in the following table.

System	Cost	Scalability	Traffic Spikes	Maintenance
System 1	High ($5.5M) but one-time	Limited (up to a few thousand hosts)	Potential issues beyond capacity	Low effort but limited customization
System 2	Moderate ($121,400/month) but high cost of opportunity	Challenging Manual scaling GCP limits	Severe issues without planning or overcapacity design	High effort Customizable
System 3	High ($236,700/month)	No issues (handled by provider)	No issues but higher costs	Moderate effort Low maintenance SaaS solution

Table 5.2: Summary of system performance in cost, scalability, traffic spike handling, and maintenance

5.2 Simulations

In this section there will be presented 3 situations to test the three systems:

- The first scenario is **normal operation**: This aims to simulate the typical day-to-day functioning, allowing us to assess which system performs best under ideal conditions.
- The second scenario is **traffic spikes**: This scenario simulates an unexpected surge in traffic, with the goal of evaluating which system exhibits greater resilience under unusually high load.
- The third scenario is **system outage**: This simulates a situation where a component, such as a database, is non-operational. The objective is to determine which system responds more effectively to the failure, focusing on both issue detection and resolution capabilities.

5.2.1 Scenario 1

- **System 1:** This system operates efficiently under normal conditions due to its model, which allows for reliable functionality within the bounds of the purchased license. Its primary strength is its cost-effectiveness for small to medium-sized enterprises, as the cost is fixed regardless of the number of machines. However, its performance might degrade as the number of monitored hosts approaches its maximum capacity. The system provides comprehensive monitoring and alerting capabilities, but customization to specific needs can be complex and require significant IT expertise.
- **System 2:** Designed for flexibility and customization, this system is well-suited for environments with variable monitoring needs. It allows for extensive tailoring to meet specific requirements, and its monthly subscription model means costs are very predictable. The system provides advanced monitoring and analysis features, which can

be fine-tuned to suit the organization's needs. However, its complexity requires skilled personnel for setup and maintenance. Overall this system also performs really well under normal conditions.

- **System 3:** As a SaaS solution, System 3 offers ease of use with minimal maintenance requirements. It includes built-in scalability and performance optimization managed by the provider. This system is ideal for organizations that prefer a managed service with predictable costs that increase with usage. The provider handles the infrastructure, allowing the organization to focus on utilizing the system's features without worrying about scalability or maintenance. This is the ideal scenario for a company with unlimited budget, as it proves to have less burden on employees than the other two.

It is no surprise that all three systems perform effectively under normal conditions, provided they are set up correctly and function without any issues.
When it comes to routine operations, the optimal choice among the three systems depends on their specific strengths and weaknesses, as well as their associated costs.

5.2.2 Scenario 2

- **System 1:** During traffic spikes, System 1 may face challenges due to its limited scalability. Although the price allows for using as many machines as needed (in the unlimited hosts price range), the system might not handle sudden surges in traffic effectively. It is essential to plan for such spikes within the system's capacity limits to avoid performance issues. Choosing this system means possibly losing control over the whole flow.

- **System 2:** Managing this system during unexpected traffic spikes can be challenging. Its need for manual scaling means that without advance notice, the system may struggle to handle sudden high loads. The cost of rapidly scaling up can be substantial, making it less suitable for environments with unpredictable traffic patterns. Effective management of traffic spikes requires careful planning and additional resources. While this system also faces difficulties with traffic spikes, it remains possible to address them, unlike System 1, which has a hard limit on scalability.

- **System 3:** System 3 excels at handling traffic spikes due to its managed scalability. The provider can adjust resources dynamically to accommodate increased loads, ensuring continuous performance. However, costs will increase with higher traffic volumes, and it is crucial to ensure that the service contract includes provisions for high usage to avoid unexpected charges.

In this scenario, System 3 clearly emerges as the best option, effortlessly managing traffic spikes since all scalability concerns are handled by the service provider. If traffic spikes are anticipated or could be a potential issue, System 3 offers superior efficiency and cost-effectiveness. System 1, on the other hand, has a scalability limit that cannot be exceeded, while System 2 would require substantial infrastructure expansion and extensive manual intervention to accommodate high loads.
However, if traffic spikes are unlikely or if the company can tolerate potential monitoring disruptions during such events, Systems 1 or 2 might still be viable alternatives.

5.2.3 Scenario 3

- **System 1:** During a system outage, System 1 depends on its built-in monitoring and alerting functionalities. The system's effectiveness in detecting and addressing issues hinges on its configuration and any supplementary support arrangements. Although it offers strong monitoring capabilities, resolving issues might be slower if the system's capacity is exceeded or if complex custom configurations are required. However, with thoughtful planning and the appropriate selection of plugins, many of these challenges can be managed or prevented.

- **System 2:** System 2 provides extensive customization options, which can be valuable for identifying and resolving system outages. Its advanced features and manual management capabilities enable tailored solutions for handling failures. However, the system's complexity may affect response speed, necessitating skilled personnel to address issues promptly. Its in-depth approach often allows for more precise identification of problems compared to System 1, including the ability to pinpoint exactly which machine caused the error, thereby facilitating quicker issue resolution.

- **System 3:** As a SaaS solution, System 3 benefits from the provider's dedicated support for managing outages. The provider is responsible for maintaining system reliability and addressing failures, which typically results in a faster and more efficient resolution. Organizations benefit from the provider's expertise and infrastructure, though they rely on the provider's responsiveness and support quality.

While all three systems perform effectively in this scenario, the last two may offer slightly faster resolution due to their more machine-specific approaches. The identification and resolution processes can be significantly expedited by utilizing tools like Sentry, which can pinpoint the exact lines of code where the error occurred, thus aiding in the swift resolution of bugs that might be causing the issue.

Chapter 6

Conclusions

6.1 Results

In conclusion, this study examined three distinct monitoring systems: the first offering an out-of-the-box solution, the second a fully customized system built on popular and powerful open-source projects, and the third utilizing the same open-source projects but offered as a SaaS (Software as a Service) solution, optimized for scalability and robustness.
Each system demonstrated unique strengths that were unmatched by the others. System 1 stood out for its simplicity in setup and maintenance, making it an ideal choice for organizations with minimal technical staff. System 2 excelled in customizability at a relatively low cost, making it suitable for companies with specific monitoring needs who prefer a solution tailored to their existing infrastructure. Finally, System 3 showcased superior scalability, making it well-suited for large-scale applications or organizations managing a high volume of users or applications.
Each system also presented certain limitations. System 1 proved less effective for highly technical and scalable environments. While System 2 offered better scalability and leveraged more modern software, it remained limited in horizontal scaling and required manual intervention for scaling, which was slow and cumbersome. Additionally, System 2 struggled to efficiently manage traffic spikes. In contrast, System 3 provided the highest level of scalability and handled traffic spikes with ease, though this came at the cost of reduced customizability and a significantly higher price point.
Ultimately, none of the systems emerged as a definitive winner. Each is best suited for different scenarios and organizational needs, depending on the priorities of the company: whether scalability, customizability, or ease of setup is the primary concern.

6.2 Future Work

The primary objective of this thesis was to explore alternative monitoring systems with a focus on scalability. While the study provided a general overview and comparison of three systems, a more in-depth analysis was beyond the scope of this work. Such an analysis would inevitably depend on specific implementations, which could significantly influence the outcomes and lead to varying results.
One promising area for future research is the integration of OpenTelemetry, which has increasingly become the industry standard for observability. OpenTelemetry offers features such as operating in push mode rather than pull, making it particularly well-suited for environments with ephemeral jobs, an approach that is becoming more prevalent in microservices

architectures. Future work could explore how systems like those discussed in this thesis (especially Systems 2 and 3) could be built or adapted using OpenTelemetry, leveraging its advanced capabilities to enhance scalability and flexibility.

Additionally, the current analysis does not include any optimization of the systems. The focus was on evaluating the systems as they are, without exploring potential optimizations that could improve performance or cost efficiency. Future research could investigate areas within each system where optimizations could be applied, such as resource allocation, infrastructure configuration, or monitoring strategies, to achieve better outcomes tailored to specific use cases. Another avenue for future work could involve exploring cost optimization strategies for Systems 2 and 3, as the current analysis is based on public, usage-based pricing. This study did not examine ways to reduce costs by identifying primary cost drivers and shifting to more affordable alternatives. It is also important to note that enterprise-level contracts often provide discounts compared to public pricing, which were not considered in this analysis. Since these discounts are highly company-specific, each organization should conduct its own cost-benefit analysis, similar to the approach shown here, but tailored to its specific financial data. By incorporating such optimizations, a more refined comparison could emerge, potentially leading to different conclusions.

Finally, a more comprehensive exploration of automation and self-healing capabilities could provide valuable insights into reducing manual intervention during outages and enhancing overall system resilience.

Bibliography

[1] M. Lopes and A. Hora, “How and why we end up with complex methods: a multi-language study,” *Empirical Software Engineering*, vol. 27, no. 5, pp. 1–29, 2022.

[2] S. Laato, M. Mäntymäki, A. Islam, and et al., “Trends and trajectories in the software industry: implications for the future of work,” *Information Systems Frontiers*, vol. 25, pp. 929–944, 2023.

[3] Statista, “Global software market size from 2012 to 2025.” `https://www.statista.com/markets/418/topic/484/software/#statistic1`, 2023. Accessed: 2024-09-21.

[4] S. Elliot, “Devops and the cost of downtime: Fortune 1000 best practice metrics quantified,” *International Data Corporation (IDC)*, 2014.

[5] Alphabet Inc., “Alphabet inc. q2 2024 earnings release,” 2024. Accessed: 2024-09-21.

[6] S. Rodrigues, “Infoworld’s best of open source software 2008.” Available: `https://www.infoworld.com/article/2311663/infoworld-announces-our-2008-best-of-open-source-awards.html`, August 2008. Last accessed: 2024-08-10.

[7] J. Gray, “Readers’ choice awards 2009.” Available: `http://www.linuxjournal.com/article/10451`, June 2009. Last accessed: 2024-08-10.

[8] “Linuxquestions.org network monitoring application of the year 2018.” Available: `https://www.nagios.com/awards/`, August 2018. Last accessed: 2024-08-10.

[9] “Nagios case studies.” Available: `https://www.nagios.com/case-studies/`. Last accessed: 2024-08-10.

[10] “Nagios core monthly downloads on sourceforge.net.” Available: `https://sourceforge.net/projects/nagios/files/stats/timeline?dates=2001-05-01%20to%202024-08-01&period=monthly`. Last accessed: 2024-08-10.

[11] “Nagios core 4 documentation.” Available: `https://assets.nagios.com/downloads/nagioscore/docs/nagioscore/4/en/plugins.html`. Last accessed: 2024-08-10.

[12] “Latvian open technologies association ”the most open solution”, 2021.” Available: `https://www.zabbix.com/pr/pr321`, February 2021. Last accessed: 2024-08-10.

[13] “April 2019 gartner peer insights customers’ choice for it infrastructure monitoring tools.” Available: `https://blog.zabbix.com/zabbix-users-have-spoken/6888/`, April 2019. Last accessed: 2024-08-10.

[14] “Prometheus documentation.” Available: `https://prometheus.io/docs/introduction/overview/`. Last accessed: 2024-08-28.

[15] "Grafana documentation." Available: `https://grafana.com/docs/grafana/latest/fundamentals/intro-to-prometheus/`. Last accessed: 2024-08-28.

[16] "Cortex documentation." Available: `https://cortexmetrics.io/docs/`. Last accessed: 2024-08-28.

[17] "Pagerduty documentation." Available: `https://support.pagerduty.com/main/docs/`. Last accessed: 2024-08-28.

[18] S. Thurgood, J. Frame, A. Lenton, C. Quinito, A. Tolchanov, and N. Trdin, "Alerting on slos," 2018. Accessed: 2024-09-21.