**IMPLEMENTING OBSERVABILITY IN A SYSTEM USING PROMETHEUS AND GRAFANA**

by

MANPREET SINGH

Submitted in partial fulfilment of the requirements for the degree of

MASTER OF ENGINEERING

Major Subject: Internetworking

at

DALHOUSIE UNIVERSITY

Halifax, Nova Scotia   
July, 2024

**©** Copyright by *Manpreet Singh, 2024*

**Dalhousie University**

**Faculty of Engineering**

**Internetworking**

The undersigned hereby certify that they have read and awarded a pass in INWK 6800 for the course/seminar entitled "Implementing Observability in A System Using Prometheus And Grafana" by Manpreet Singh in partial fulfilment of the requirements for the degree of Master of Engineering.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Dr. Muhammad Raza

**DALHOUSIE UNIVERSITY**

**INTERNETWORKING PROGRAM**

**AUTHORITY TO DISTRIBUTE REPORT**

Title:

IMPLEMENTING OBSERVABILITY IN A SYSTEM USING PROMETHEUS AND GRAFANA

The Internetworking Program may make available or authorise others to make available individual photo/microfilm or soft copies of this report without restrictions after 26-July-2024.

The author attests that permission has been obtained for the use of any copyrighted material appearing in this report and that all such use is clearly acknowledged.

Full Name of Author: Manpreet Singh

Signature of Author:

Date: July 26, 2024

**Table of Contents**

LIST OF FIGURES ………………………………………………………………………………………...v

**ACKNOWLEDGEMENTS ………………………………………………………………………………vi**

**EXECUTIVE SUMMARY ………………………………………………………………………………vii**

[1 INTRODUCTION 1](#_Toc172643147)

[1.1What Is Observability 2](#_Toc172643148)

[1.2 Purpose Of The Project 3](#_Toc172643149)

[1.3 Implementing Observability On Localhost 3](#_Toc172643150)

[1.4 Outline 4](#_Toc172643151)

[2 COMPONENTS OF PROMETHEUS 5](#_Toc172643152)

[2.1 Prometheus Server 6](#_Toc172643153)

[2.2 PromQL 11](#_Toc172643154)

[2.3 Time-Series Database 12](#_Toc172643155)

[2.4 Exporters 13](#_Toc172643156)

[2.5 AlertManager 16](#_Toc172643157)

[2.6 Integrating Grafana for Data Visualization 22](#_Toc172643158)

[3 INSTRUMENTING APPLICATION CODE FOR PROMETHEUS 25](#_Toc172643159)

[4 FUTURE SCOPE OF THE PROJECT 26](#_Toc172643160)

[5 REFERENCES 27](#_Toc172643161)

LIST OF FIGURES

Figure 2.0 Prometheus Architecture …………………………………………………….. 3

Figure 2.1 Prometheus Service Deployed On LocalHost ……………………………….. 4

Figure 2.2 Prometheus UI ……………………………………………………………….. 5

Figure 2.3 Prometheus Metrics Exposed ………………………………………………… 7

Figure 2.4 Prometheus.yaml ……………………………………………………………... 7

Figure 2.5 Sample PromQL Query ……………………………………………………… 8

Figure 2.6 Time-Series Databases ……………………………………………………….. 8

Figure 2.7 NodeExporter Exposed on Port 9100 On LocalHost ………………………… 9

Figure 2.8 NodeExporter UI ……………………………………………………………. 10

Figure 2.9 AlertManager Deployed …………………………………………………….. 11

Figure 2.10 AlertManager UI …………………………………………………………... 12

Figure 2.11 Alert.yaml Configuration File ……………………………………………... 13

Figure 2.12 Configuring SMTP For AlertGeneration ………………………………….. 14

Figure 2.13 NodeHighCPUUsage Alert Triggered To Gmail System …………………. 15

Figure 2.14 Grafana Tool Exposing The NodeExporter Metrics Of LocalHost ……….. 16

Figure 2.15 Grafana Service Deployed On LocalHost …………………………………. 17

Figure 3.1 Sample Prometheus GoLang Library ……………………………………….. 18

ACKNOWLEDGEMENTS

I extend my heartfelt gratitude to instructor **Dr. Muhammad Raza** for his invaluable guidance and unwavering support throughout my journey with this research report. His profound insights and patient mentorship have been instrumental in shaping the concept and structure of this work.

I would also like to express my sincere appreciation to **Dr. Bill Robertson**, Program Director, and **Ms. Shelley Caines**, Student Advisor & Director of the Internetworking program at Dalhousie University. Their visionary leadership and meticulous planning have been crucial in orchestrating the Internetworking system program and ensuring its success. Without their dedicated coordination, this achievement would not have been possible.

Furthermore, my deepest thanks go to all the instructors and staff of this course for their continuous support, encouragement, and guidance during the entire duration of the program. Their commitment to excellence has been a constant source of inspiration.

EXECUTIVE SUMMARY

Implementing observability into a system involves creating the capability to measure, monitor, and understand the internal state of the system. This implementation process is multifaceted, encompassing the definition of key metrics, the storage of relevant information in a time series database, and the establishment of dashboards for clear visualization. For instance, tools like Grafana are utilized for visualization, while Prometheus Alertmanager is employed for configuring alerts based on defined threshold or metric parameters. Metrics are numerical values that can be measured and tracked over time. They provide objective data points that reflect the state of the system. Observability holds a significant space in the field of platform engineering. Without observability, diagnosing and troubleshooting issues quickly becomes challenging. When a problem occurs, pinpointing the root cause can be difficult, leading to increased downtime and frustration. The absence of metrics, logs, and traces results in limited visibility into the system’s behavior and performance, making it hard to understand what went wrong.The objective of the project is to delve into the workings of Prometheus and Grafana tools by scraping the metrics of the localhost where the system is deployed and generating alerts. These alerts are then used to notify a Gmail system when a threshold condition is reached. The workflow starts by identifying key performance indicators (KPIs) and system metrics that need to be monitored. These metrics are collected and stored using Prometheus, which scrapes the data from various endpoints. Once the data is collected, it is visualized using Grafana, a powerful open-source visualization tool. Grafana allows the creation of detailed and customizable dashboards that provide insights into the system’s performance and health. These dashboards help in identifying trends, spotting anomalies, and understanding the overall state of the system at a glance. Prometheus Alertmanager is configured to set up alerts based on predefined threshold conditions. For example, if CPU usage exceeds a certain percentage or if memory consumption reaches a critical level, an alert is triggered. These alerts are then integrated with Gmail to notify the relevant engineers or stakeholders. This immediate notification system ensures that issues are addressed promptly, reducing downtime and maintaining system reliability. Once an alert is generated, the engineer investigates the issue by diving deep into the alert details. This process involves analyzing the metrics, logs, and traces to understand the root cause of the problem. Based on the findings, the engineer works towards implementing workarounds or permanently resolving the issue. This might include optimizing resource usage, fixing bugs, or making configuration changes to improve system performance. In summary, implementing observability is crucial for maintaining robust and high-performing systems. By using tools like Prometheus and Grafana, teams can gain comprehensive visibility into their systems, set up effective monitoring and alerting mechanisms, and ensure quick resolution of issues. This not only enhances system reliability but also empowers engineering teams to achieve operational excellence.

# INTRODUCTION

Prometheus is a robust open-source system and time-series database specifically designed for monitoring and observability. Developed initially by SoundCloud, Prometheus has emerged as a cornerstone in the landscape of cloud-native computing and DevOps practices. It excels in collecting metrics from diverse sources, empowering teams to monitor everything from infrastructure performance to application behavior in real-time.

With its multidimensional data model, powerful query language (PromQL), and efficient storage, Prometheus enables proactive monitoring and troubleshooting across dynamic, distributed environments. Its ecosystem thrives on integrations with a wide array of software systems, fostering a community-driven approach that continuously evolves to meet the growing demands of scalability, reliability, and actionable insights in today's complex technological landscapes.

Prometheus supports a pull-based model, where it scrapes metrics from configured endpoints, ensuring that it can easily adapt to various monitoring needs. It includes a robust alerting system that works seamlessly with tools like Alertmanager, allowing teams to set up precise alerting rules and manage notifications effectively. This flexibility ensures that critical issues are promptly addressed, minimizing downtime and enhancing system resilience.

One of the defining features of Prometheus is its ability to handle high cardinality data, making it well-suited for modern, microservices-based architectures where the number of unique metric series can grow rapidly. This capability, combined with its efficient time-series database, allows Prometheus to scale horizontally, providing reliable performance even as the volume of monitored data increases.

Prometheus also integrates smoothly with Grafana, a leading open-source visualization tool, enabling users to create detailed and informative dashboards that offer deep insights into system performance and health. This integration is pivotal for data-driven decision-making, as it allows teams to visualize trends, identify anomalies, and correlate metrics from different parts of the system.

Its role in the DevOps toolkit is indispensable, providing the insights and capabilities needed to maintain robust, high-performing systems in an era where technology is both complex and critical to business success.

## What Is Observability

Observability in a system refers to the ability to understand and diagnose what is happening inside the system based on its external outputs or behaviors. It is a critical aspect of system design and operation, particularly in complex and distributed systems. Here are several key reasons why observability is essential:

1. Diagnosing Issues: In any system, problems can arise unexpectedly. Observability allows engineers and operators to monitor the system's behavior in real-time or retrospectively. This monitoring helps identify the root cause of issues quickly, leading to faster resolutions and minimizing downtime.
2. Understanding Performance: Observability provides insights into how the system is performing under normal conditions and during peak loads or stress. This visibility helps in optimizing performance and resource allocation, ensuring efficient use of system resources.
3. Monitoring User Experience: For systems that interact directly with users (such as web applications or customer service platforms), observability helps track and understand user interactions and experiences. This information is crucial for improving user satisfaction and addressing usability issues.
4. Analyzing Trends and Patterns: By collecting and analyzing observational data over time, patterns and trends can be identified. These insights enable proactive decision-making, such as capacity planning, feature enhancements, or detecting potential security threats.
5. Debugging and Iterative Development: During the development phase, observability tools aid in debugging code and verifying new features. Developers can monitor how changes affect the system and validate performance against expected outcomes.

## 1.2 Purpose Of The Project

Since platform monitoring is an imperative system that helps us track and optimize the system, it plays a crucial role in maintaining the health and performance of our infrastructure. By continuously observing various components, such as servers, databases, applications, and network devices, we can gather valuable metrics and insights. These metrics include CPU usage, memory consumption, disk I/O, network traffic, and application performance. With this data, we can quickly identify potential issues, bottlenecks, and inefficiencies. Furthermore, platform monitoring enables proactive maintenance by alerting us to problems before they escalate into critical failures. It also provides the foundation for performance optimization, allowing us to fine-tune our systems, improve resource allocation, and ensure a seamless user experience. In essence, robust platform monitoring is essential for the reliable and efficient operation of any modern IT environment.

## 1.3 Implementing Observability On Localhost

In order to execute the project, we have deployed a stack of services on the local host. This setup involves configuring and running multiple interconnected services that work together to achieve our project goals. We have carefully chosen a suite of tools and platforms that ensure reliability, scalability, and performance. Among these tools, Prometheus stands out as a crucial component for monitoring and alerting. The official documentation provided by Prometheus was referenced extensively to understand the depth and nuances of the system. This documentation has been invaluable in guiding me through the complexities of setting up and configuring Prometheus for effective monitoring and alerting.

The decision to use Prometheus was influenced by its robust capabilities in time-series data collection and querying, as well as its powerful alerting mechanisms. To integrate Prometheus seamlessly with our stack, we had to ensure compatibility with various other services such as Grafana for visualization, Alertmanager for handling alerts, and several exporters to collect metrics from different parts of our system. Each of these components required meticulous configuration to ensure they functioned harmoniously.

The extensive documentation helped us navigate these challenges, providing detailed explanations and examples that were directly applicable to our use case.

## 1.4 Outline

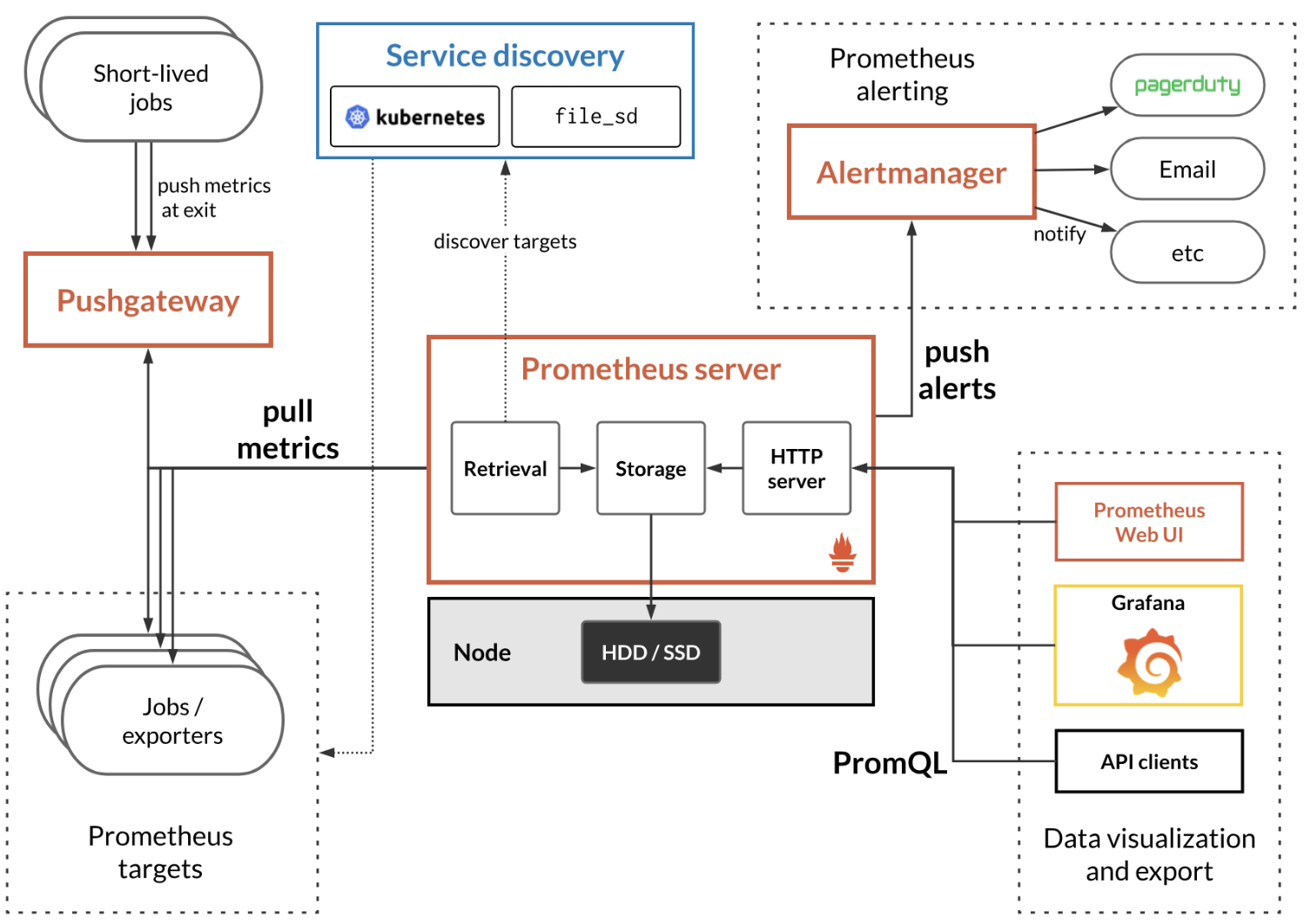
Chapter 2explores the architecture and implementation of Prometheus by doing a comprehensive deep dive into the list of components like AlertManager, Time-Series Database, Exporters etc. and data visualization tools that are supported by Prometheus

Chapter 3 explores Instrumentation of application code in Prometheus which refers to the process of adding specific code to your application to collect and expose metrics that can be scraped by Prometheus.

Chapter 5 explains the future scope of the project where the implementation is deployed into a Kubernetes infrastructure as this platform works natively with Kubernetes

# COMPONENTS OF PROMETHEUS

The emphasis of this section is to understand the core components of the Prometheus tool. We have divided this section into subsections to describe the different components Prometheus provides to implement observability. Below figure 2.0 shows the general architecture of Prometheus.



**Figure 2.0 Prometheus Architecture (reference: google)**

Prometheus actively scrapes data, stores it, and supports queries, graphs, and alerts, as well as provides endpoints to other API consumers like [Grafana](https://grafana.com/). It does all of this via. the following components:

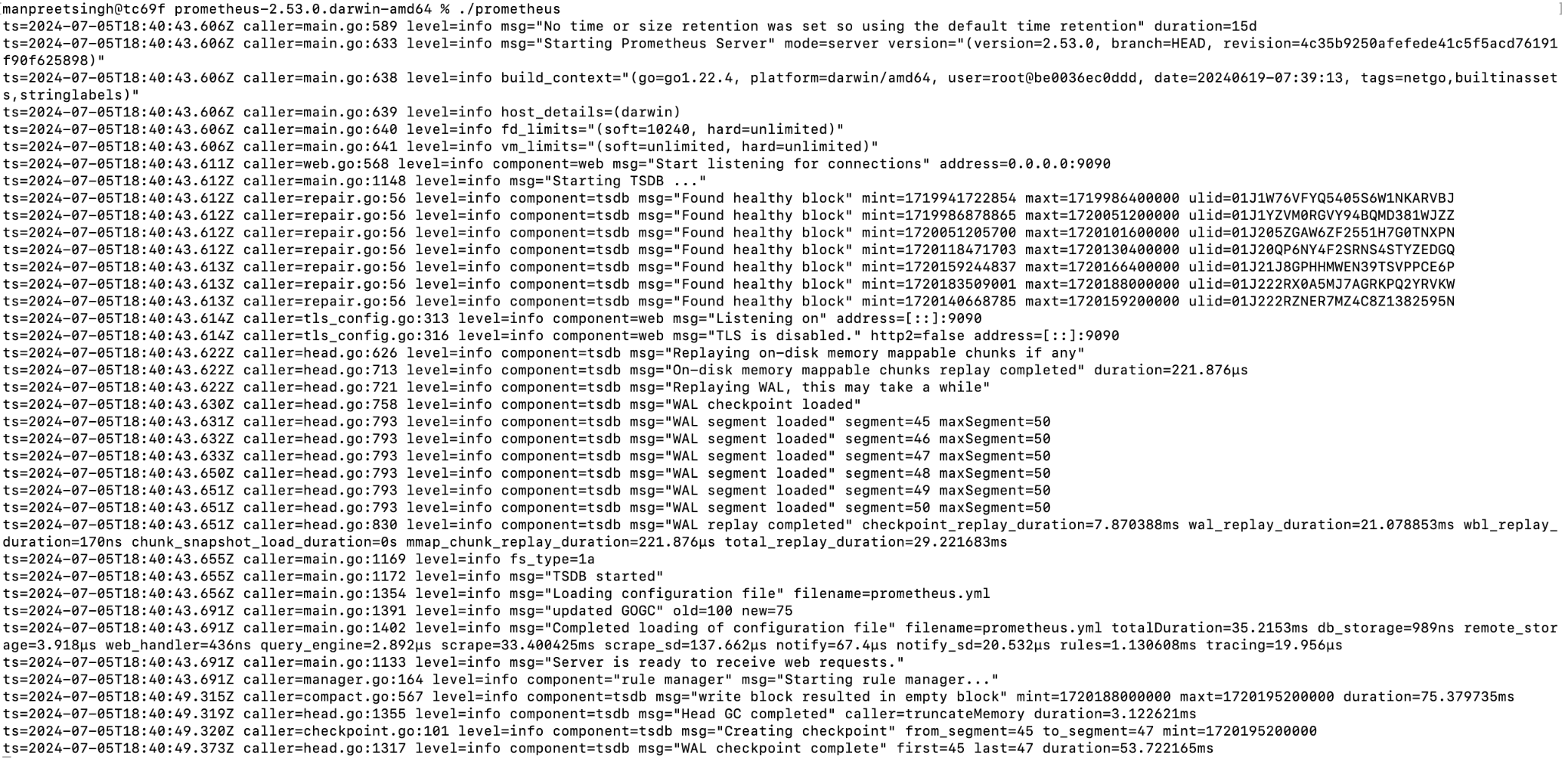
1. [Prometheus Server](https://github.com/prometheus/prometheus) – scraping and storing these events, when fired, as time series data
2. [Client Libraries](https://prometheus.io/docs/instrumenting/clientlibs/) – instrumenting application code (for generating events)
3. [PushGateway](https://github.com/prometheus/pushgateway) – supporting short-lived data import jobs eg. cronjobs
4. E[xporters](https://prometheus.io/docs/instrumenting/exporters/) – exporting to services like HAProxy, StatsD, Graphite, etc.
5. [AlertManager](https://github.com/prometheus/alertmanager) – managing alerts

## Prometheus Server

Prometheus Server stands as the cornerstone of the Prometheus observability ecosystem, pivotal in modern cloud-native monitoring and alerting practices. Functioning as both collector and storage engine, it efficiently gathers time-series metrics from diverse targets via. HTTP endpoints, employing a multi-dimensional data model with labels for flexible querying and aggregation. Supported by PromQL, its powerful query language, Prometheus Server enables detailed analysis and real-time monitoring of application and infrastructure performance. With robust alerting rules and integration with Alertmanager for streamlined alert handling, it empowers proactive incident response. Its scalable architecture, coupled with support for various service discovery mechanisms and exporters, ensures adaptability to dynamic environments.

Prometheus Server provides a built-in web-based user interface (UI) for exploring metrics, running queries, and visualizing data. Additionally, it exposes a comprehensive HTTP API that allows programmatic access for querying metrics, managing alerting rules, and more.

The Prometheus server runs on port 9090. Below figure 2.1 shows prometheus deployed on the localhost.



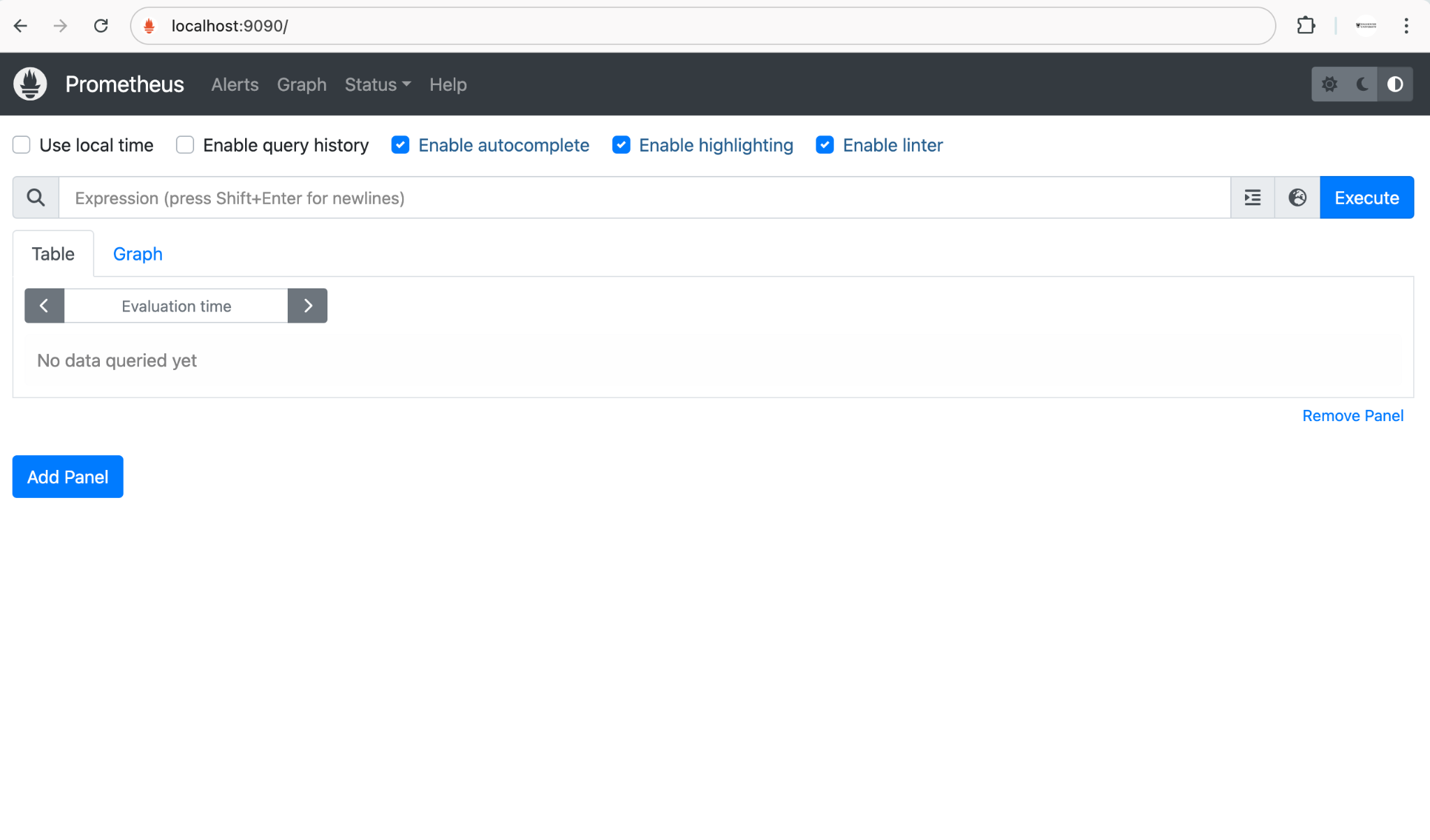
**Figure 2.1 Prometheus Service Deployed on LocalHost**

Prometheus UI:

Prometheus UI is a crucial component of the Prometheus monitoring system,

providing a powerful and intuitive way to interact with the collected time-series data. The UI is web-based and accessible via a browser, typically running on port 9090 by default. This interface allows users to explore metrics, execute queries, and visualize data in real-time, making it an essential tool for monitoring and troubleshooting purposes. At its core, the Prometheus UI offers a query editor where users can write and run PromQL (Prometheus Query Language) queries. PromQL is a flexible and expressive language designed specifically for querying the multidimensional data model used by Prometheus. Through the query editor, users can retrieve specific metrics, aggregate data, and create complex expressions to gain insights into their system’s performance and behavior. The UI also features a built-in graphing tool that allows users to visualize query results. These visualizations can range from simple line graphs to more complex representations, enabling users to track metrics over time, identify trends, and spot anomalies. This graphical representation is crucial for understanding the state of the system at a glance and for making informed decisions based on the data.

Users can see which targets (i.e., monitored instances) are being scrapped for metrics and can troubleshoot any issues related to target availability or data collection. Below figure 2.2 shows us the Prometheus UI.

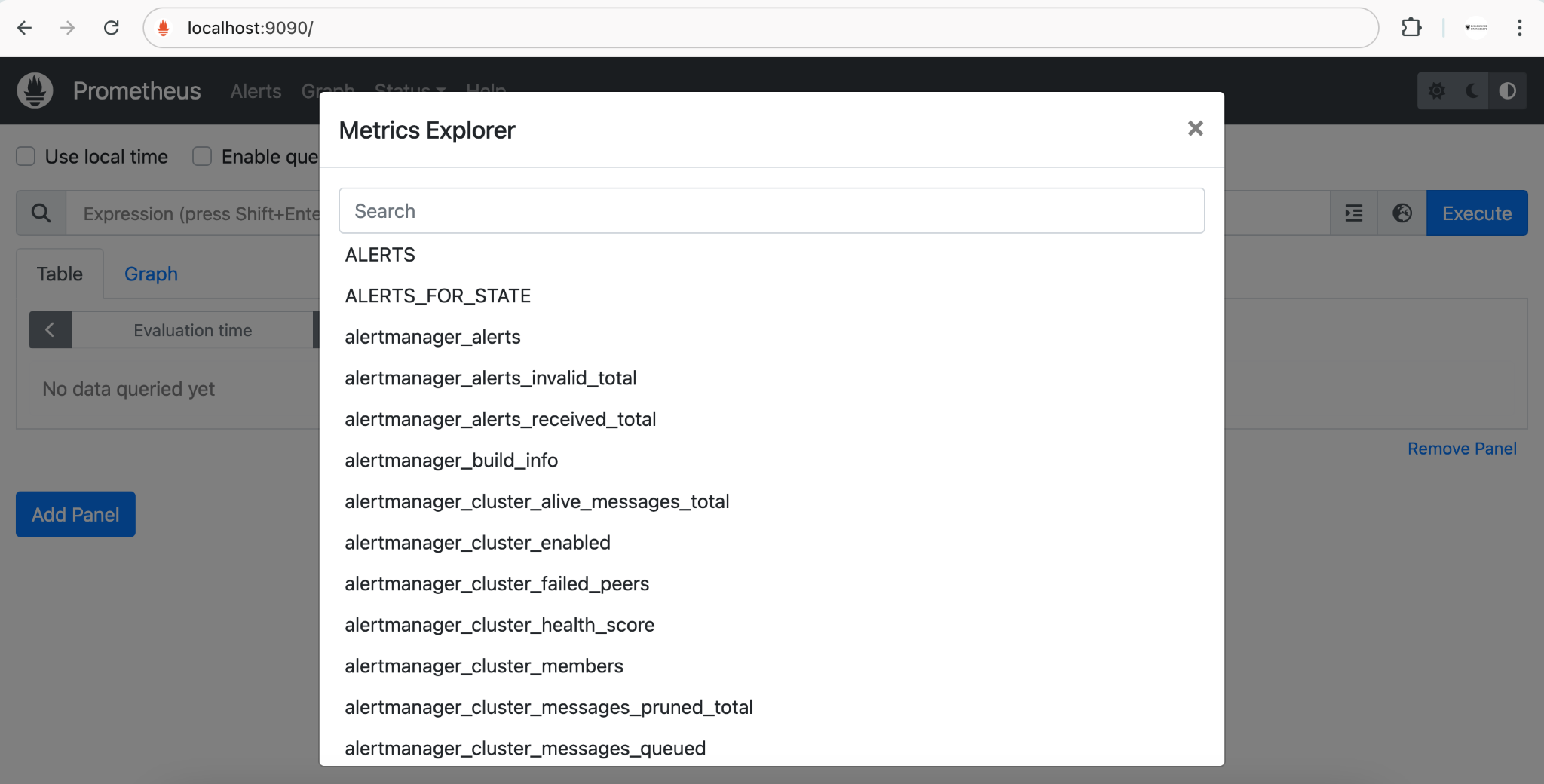


**Figure 2.2 Prometheus UI**

Metrics Exposed from Prometheus:

Metrics exposed from Prometheus are fundamental to its capability as a comprehensive monitoring and alerting system. These metrics encompass a wide range of data points that provide insights into the performance, health, and behavior of various components within an IT infrastructure. Prometheus collects these metrics from targets through HTTP endpoints, where each metric is uniquely identified by a name and a set of key-value pairs called labels. This multidimensional data model allows for powerful and flexible querying and aggregation. The metrics are categorized into four primary types: counters, gauges, histograms, and summaries. Counters represent cumulative values that only increase over time, such as the number of requests received by a server. Gauges, on the other hand, can go up and down and are used to measure values like current memory usage or the number of active connections.

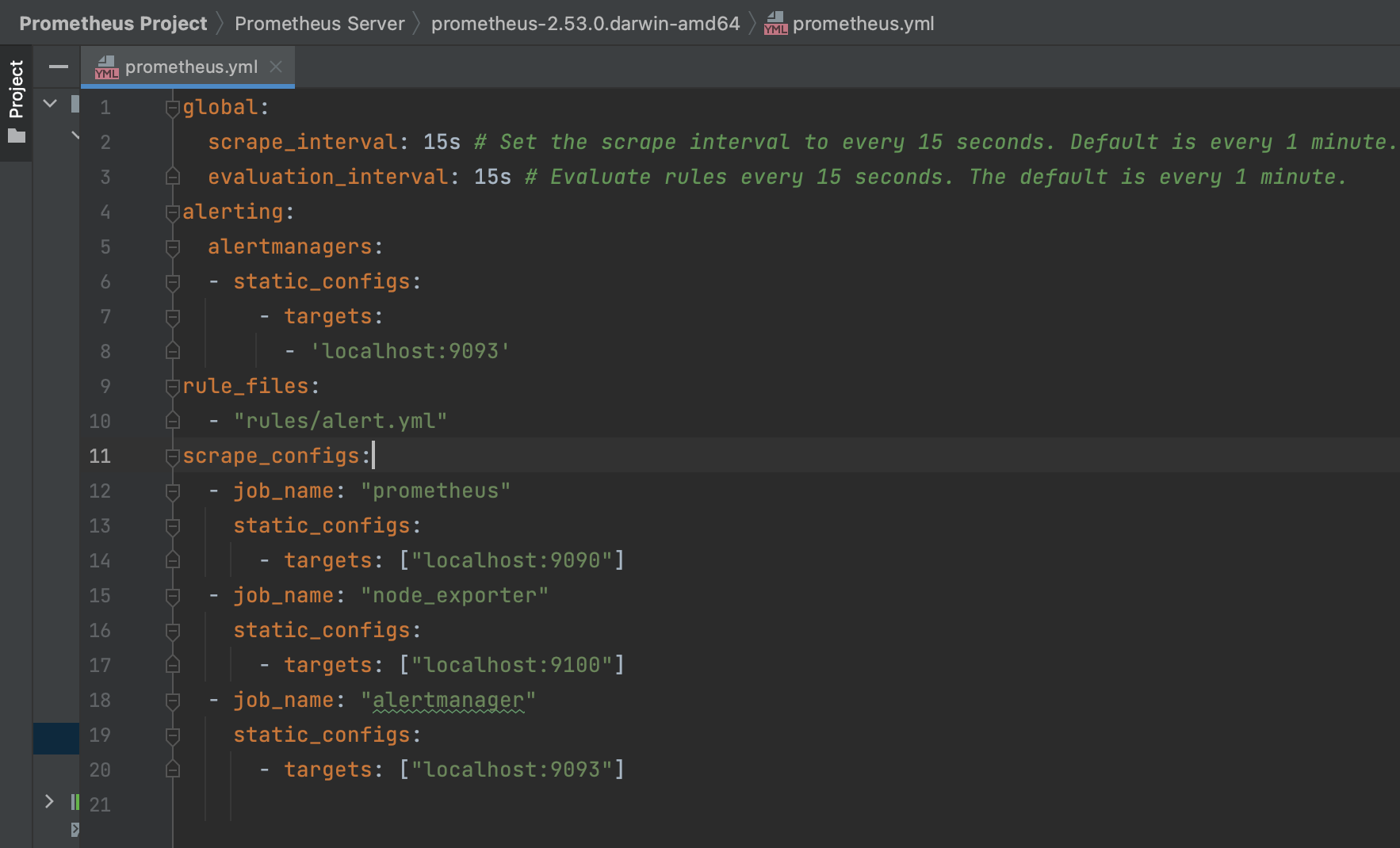
The richness of the metrics exposed by Prometheus extends beyond basic infrastructure monitoring. It includes application-level metrics, business metrics, and even custom metrics defined by the users. For instance, application developers can instrument their code to expose metrics about the internal workings of their applications, such as the number of transactions processed, error rates, or the duration of specific operations. These application-level metrics are invaluable for gaining insights into the operational aspects of the software and for detecting performance bottlenecks or anomalous behaviors. Below figure 2.3 shows the metrics exposed from Prometheus.



**Figure 2.3 Prometheus Metrics Exposed**

Prometheus Server Configuration File:

The Prometheus server configuration file, typically named prometheus.yml, is a crucial component that dictates how Prometheus operates, what data it collects, and how it interacts with various targets and services. This configuration file, written in YAML, is highly flexible and allows users to define multiple scrape jobs, alerting rules, and service discovery mechanisms. At its core, the prometheus.yml file contains global configurations that set the default behavior for all scrape jobs, such as the default scrape interval and evaluation interval, which determine how frequently Prometheus collects data and evaluates rules, respectively. Below figure 2.4 shows us the Prometheus configuration file.



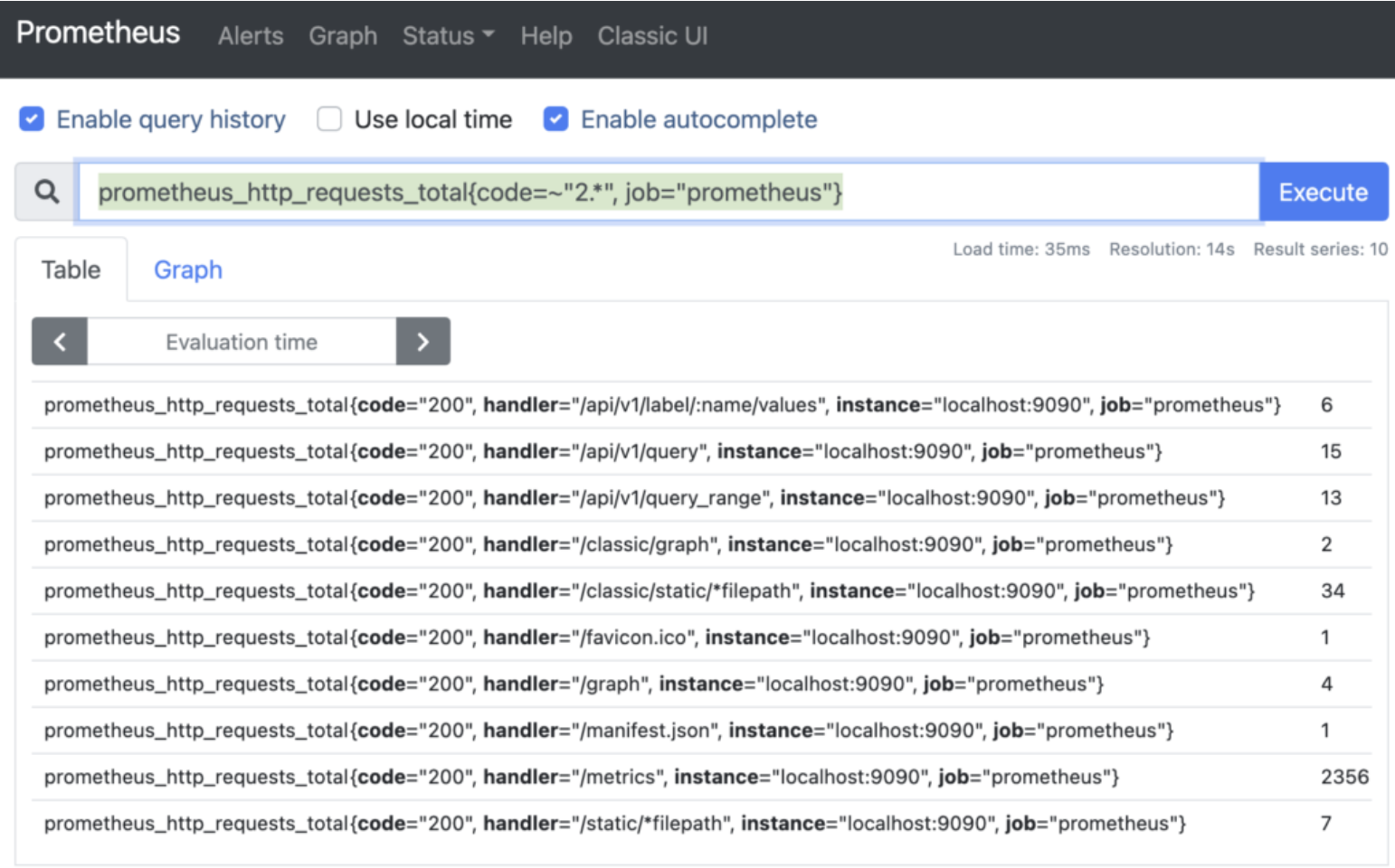
**Figure 2.4 Prometheus.yaml**

## PromQL

PromQL or Prometheus Query Language is a powerful query language designed for working with Prometheus data. It allows users to write expressive queries to retrieve, manipulate, and aggregate time-series data based on specific criteria

It is important to reveal all the details of the problem to the reader. This chapter is used to make the reader aware of all the nuances of the problem so that any subsequent trade-offs or measurements are justifiable in the context of the problem.

Below figure 2.5 is a sample PromQL query returning all values with the 200-response code from the Prometheus server



**Figure 2.5 Sample PromQL Query**

## Time-Series Database

Prometheus Server maintains a local, efficient time-series database optimized for storing metrics collected from various targets. This database structure allows for high write throughput and efficient querying of historical data. Below figure 2.6 are sample examples of Time-Series databases.



**Figure 2.6 Time-Series Databases (reference: google)**

## Exporters

Exporters in Prometheus play a pivotal role in extending the monitoring capabilities of the Prometheus ecosystem by enabling the collection of metrics from a diverse array of systems and services that do not natively expose Prometheus-compatible metrics. These exporters act as intermediaries, translating metrics from various formats or protocols into a format Prometheus can scrape and store in its time-series database. They facilitate seamless integration with a wide range of systems, including databases (such as MySQL, PostgreSQL), web servers (like Apache and Nginx), message brokers (such as RabbitMQ and Kafka), cloud platforms (like AWS and Google Cloud), and numerous other applications and services crucial to modern IT infrastructures.

Each exporter is specifically designed to understand the metrics format of the system it monitors and exposes these metrics via an HTTP endpoint typically located at /metrics. Prometheus scrapes these endpoints at configured intervals, allowing it to collect metrics on resource utilization, performance indicators, operational metrics, and more. For example, the Node Exporter is widely used to collect system-level metrics such as CPU usage, memory consumption, disk I/O, and network statistics. These metrics are crucial for understanding the health and performance of individual servers and the overall infrastructure.

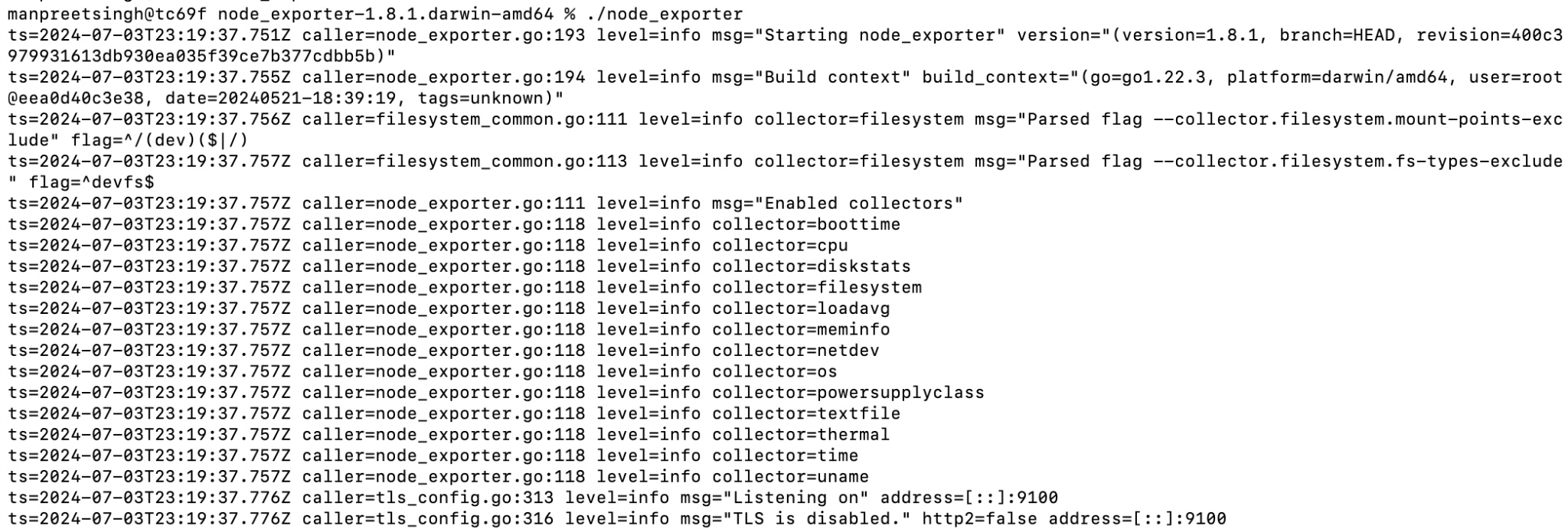
The Blackbox Exporter, on the other hand, is designed for probing endpoints via various protocols such as HTTP, ICMP, TCP, and DNS. It enables users to monitor the availability and responsiveness of web services, APIs, and network endpoints by performing synthetic checks and measuring response times. This is particularly useful for ensuring the reliability of critical services and detecting issues that might affect user experience.

Other popular exporters include the MySQL Exporter, which gathers metrics from MySQL databases such as query performance, connection counts, and replication status, and the Kafka Exporter, which collects metrics from Apache Kafka clusters, including broker performance, topic statistics, and consumer lag. The integration with cloud platforms is equally important, with exporters available for AWS, Google Cloud, and other cloud services, allowing for the monitoring of cloud resource utilization, billing metrics, and service health.

In addition to these, there are numerous other exporters available, each tailored to specific applications and services, making Prometheus a highly versatile and powerful monitoring solution. By leveraging these exporters, organizations can gain comprehensive visibility into their entire IT infrastructure, ensuring that all components, from the underlying hardware to the applications running on top, are monitored and optimized for peak performance.

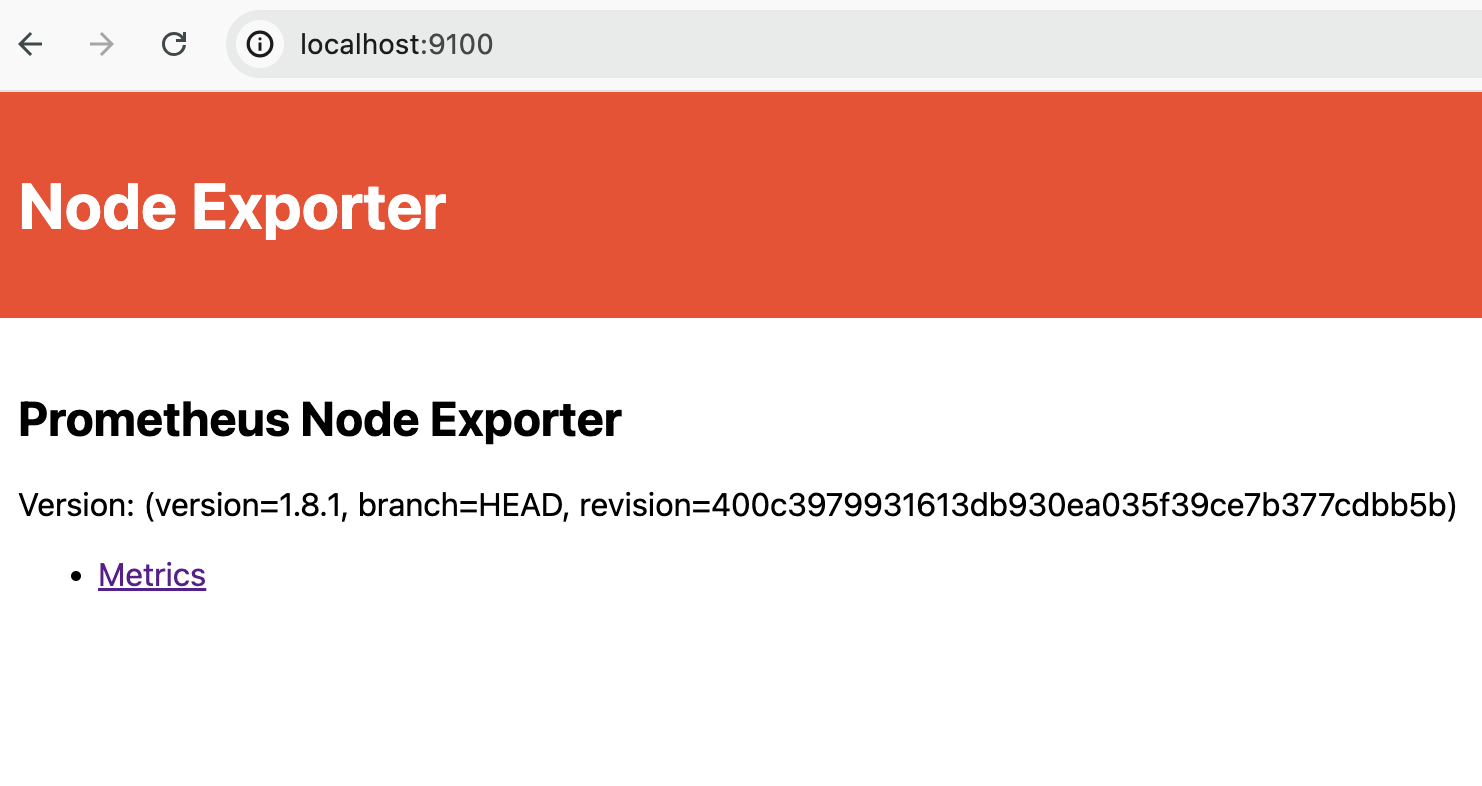
In order to deploy NodeExporter on our localhost, I downloaded the exporter libraries from the official documentation page and executed it from the path where I had placed the files for Prometheus.

Below figure 2.7 shows NodeExporter deployed on localhost.



**Figure 2.7 NodeExporter Exposed on Port 9100 On LocalHost**

Below figure 2.8 displays ‘Metrics’ which when clicked would display a list of all the metrics which are exposed from the NodeExporter thereby exposing time-series data points of various metrics of the localhost



**Figure 2.8 NodeExporter UI**

## AlertManager

Prometheus Server supports alerting based on predefined rules and thresholds. When conditions specified in alerting rules are met, Prometheus triggers alerts. These alerts

are then managed and routed by Alertmanager, a separate component closely integrated with Prometheus Server. This component runs on port 9093 and has its own configuration file to configure various alerts and send the alerts to an external system.

Key Features Of AlertManager:

1. Alert Grouping: Alertmanager can group similar alerts into a single notification. This is particularly useful for reducing noise and ensuring that users are not overwhelmed by a flood of individual alerts. Alerts can be grouped based on their labels, such as job, instance, or any custom labels.
2. Alert Deduplication: It automatically deduplicates alerts to ensure that the same alert is not sent multiple times. This helps in avoiding redundant notifications and makes sure that only unique alerts are sent to the recipients.
3. Silencing: Silences are temporary muting of alerts. This is useful for maintenance windows or known issues that are being resolved. Silences can be created using specific matchers for alert labels, allowing fine-grained control over which alerts are silenced.
4. Inhibition: Inhibition is a mechanism to suppress notifications for certain alerts if other alerts are already firing. This helps in managing alert dependencies. For example, if a high-level alert is firing, it can suppress lower-level alerts that are likely consequences of the high-level issue.
5. Routing: Alertmanager allows complex routing based on alert labels. Users can define routes that match specific labels and forward alerts to different receivers. This enables sending alerts to different teams or systems based on the alert context

Keeping the scope of the project in mind, I have configured gmail as the external system receiving the alerts when localhost’s CPU utilization crosses 20% utilization. Below figure 2.9 shows the AlertManager running on port 9093 on localhost



**Figure 2.9 AlertManager Deployed**

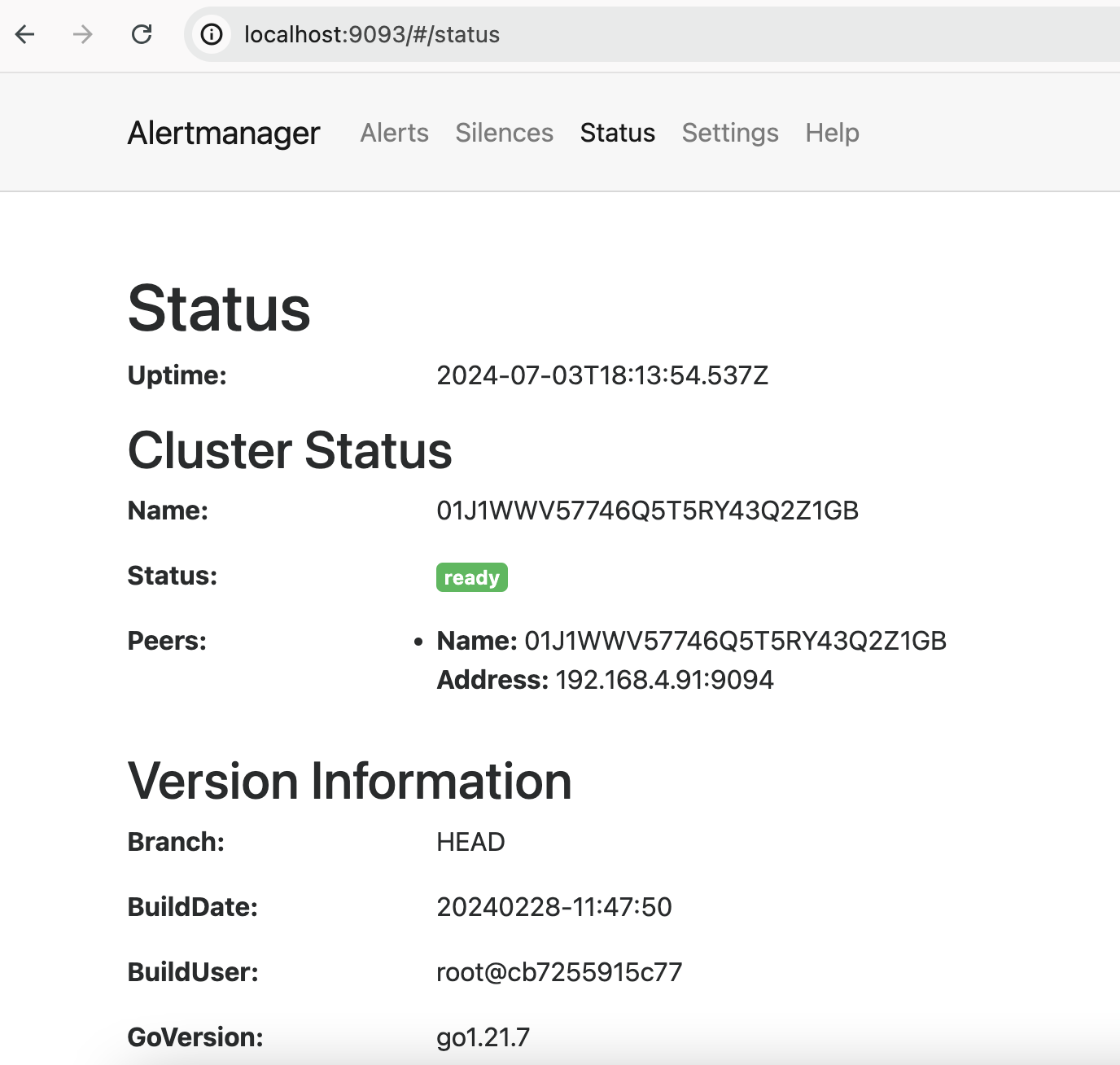
AlertManagerUI:

The AlertManager UI is a web-based interface provided by Alertmanager, a component of the Prometheus monitoring system responsible for managing alerts. The UI offers a user-friendly way to visualize, manage, and interact with alerts generated by Prometheus. Here’s a detailed look at the features and functionalities of the AlertManager UI:

Key Features Of AlertManagerUI:

1. Alert Visualization: The UI provides a comprehensive view of all active alerts. Each alert is displayed with its relevant details, including alert name, severity, labels, annotations, and the time it was last fired. This helps users quickly understand the current state of their monitoring system
2. Grouping and Filtering: Alerts can be grouped and filtered based on various criteria such as labels, severity, and status. This makes it easy to focus on specific types of alerts or alerts related to particular services or instances
3. Silences: The UI allows users to create and manage silences. Silences are used to temporarily mute alerts during maintenance windows or when dealing with known issues. Users can specify matchers for alert labels to define which alerts should be silenced and set a duration for the silence

Below figure 2.10 shows us the status page of the AlertManager deployed on port 9093 on localhost

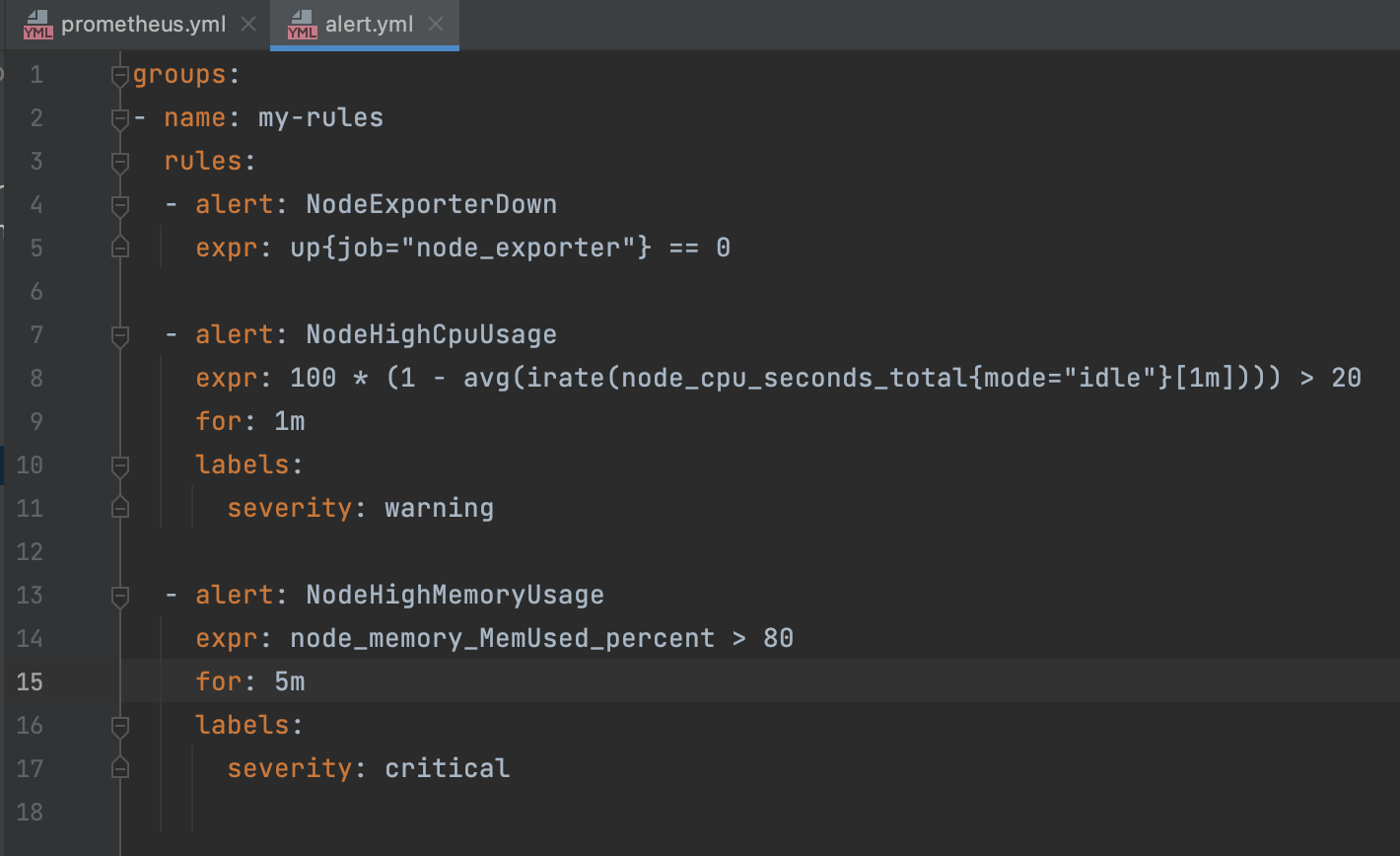


**Figure 2.10 AlertManager UI**

Alerts Configured Referring To The NodeExporter Metrics:

Configuring alerts based on Node Exporter metrics is a common practice to monitor the health and performance of your servers. By setting up alerts on these metrics, you can proactively identify and address potential issues before they impact your systems.

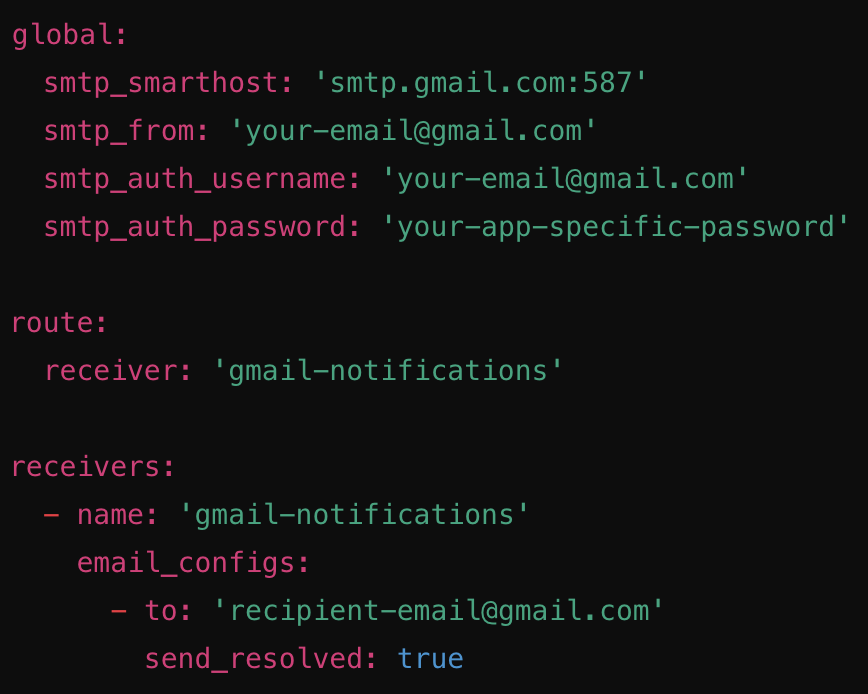
Below figure 2.11 shows us the alert.yml file which is pointed into the Prometheus.yaml file so prometheus can fire alerts to AlertManager



**Figure 2.11 Alert.yaml Configuration File**

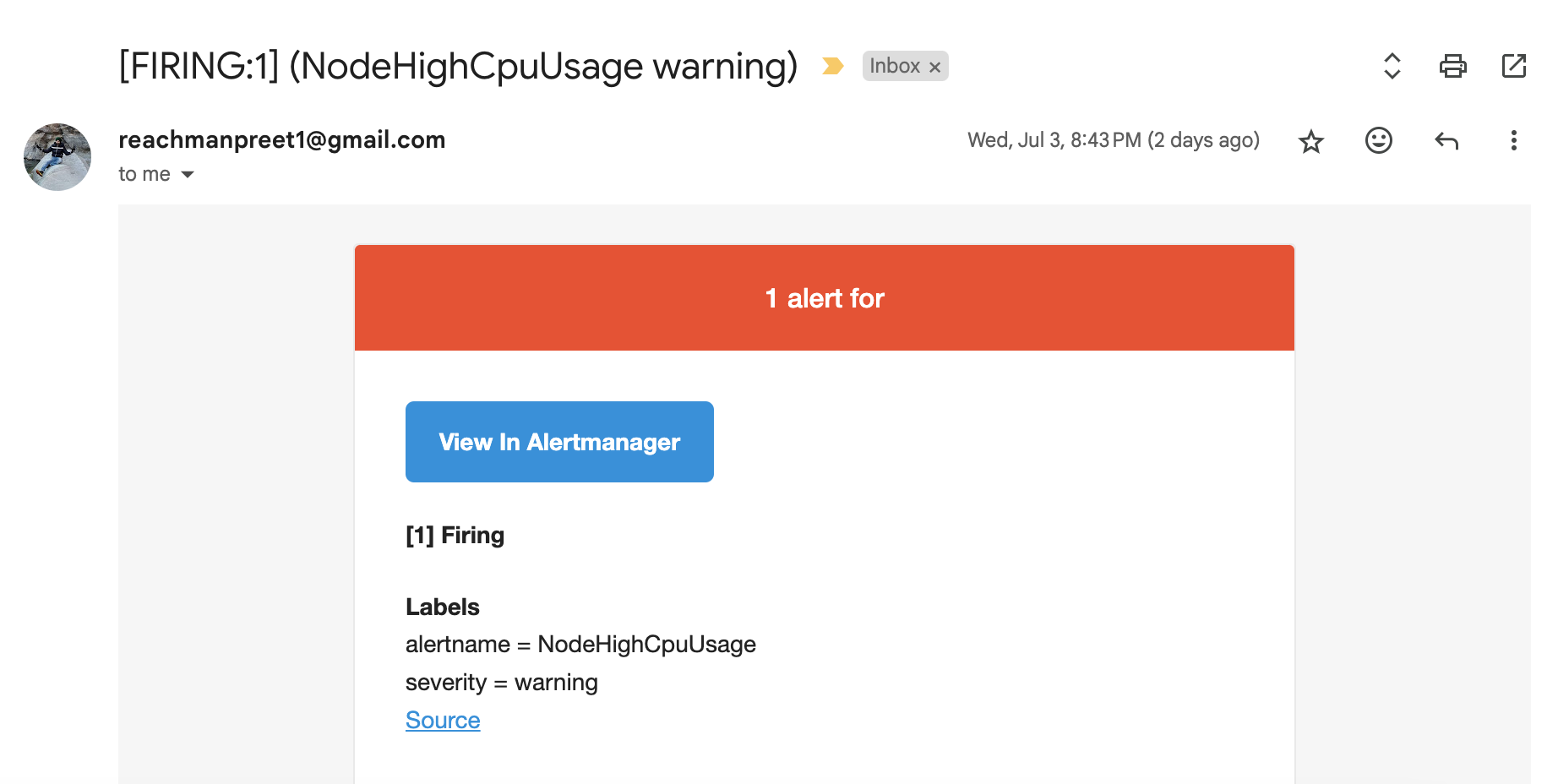
Alerts Triggered To The Gmail System From AlertManager:

Below figure 2.12 shows us the code required to configure SMTP for the AlertManager so the alerts can be triggered to the Gmail system



**Figure 2.12 Configuring SMTP For AlertGeneration**

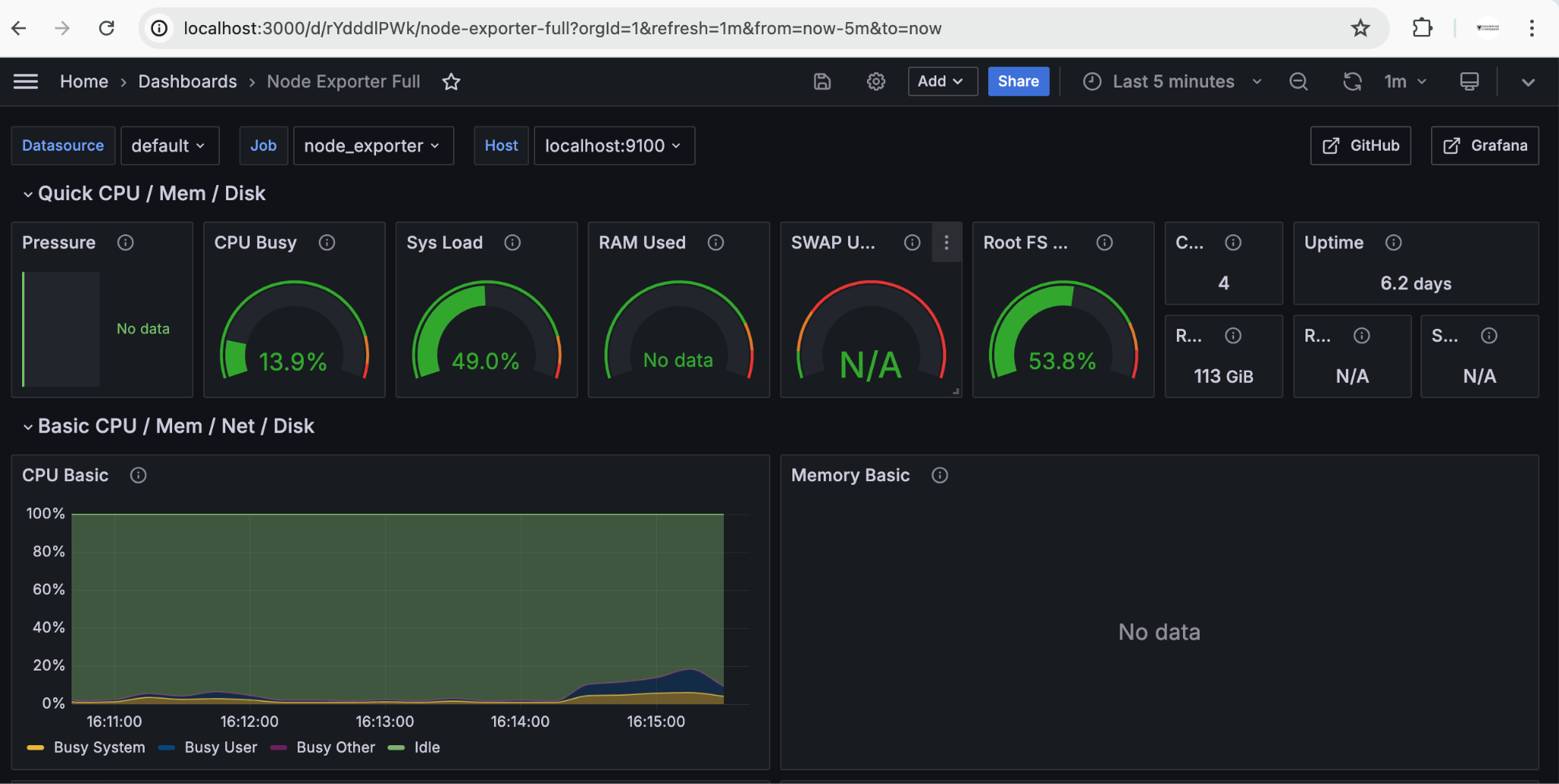
Below Figure 2.13 shows AlertManager triggering an alert to the Gmail system for High CPU usage



**Figure 2.13 NodeHighCPUUsage Alert Triggered To Gmail System**

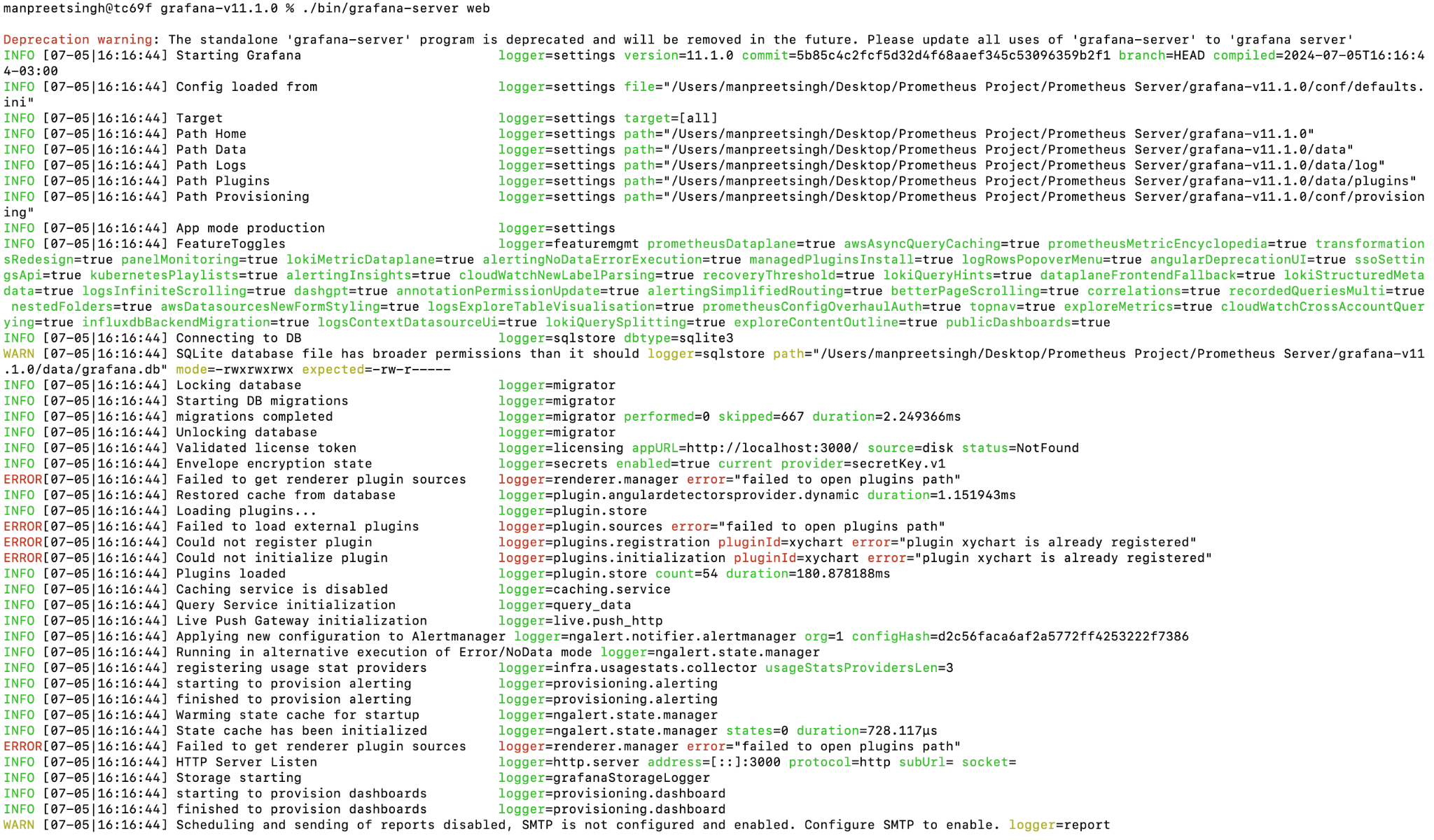
## Integrating Grafana for Data Visualization

Integrating Grafana with Prometheus for data visualization involves several key steps that enable you to effectively monitor and analyze your application and infrastructure metrics. First, you need to install Prometheus by downloading it from the official website, configuring the prometheus.yml file to specify your scrape targets, such as your application endpoints, and then running Prometheus to begin collecting metrics at defined intervals. Next, you install Grafana, which is also downloadable from its official site, and start the server, typically accessed at http://localhost:3000 with default credentials. Once Grafana is up and running, you navigate to the Data Sources section and add Prometheus as a data source by entering its URL (usually http://localhost:9090) and testing the connection to ensure everything is set up correctly. After successfully adding Prometheus, you can create a new dashboard by adding panels where you can visualize your data through various graphs and charts; here, you’ll write PromQL queries to fetch specific metrics, such as system uptime or resource utilization. Grafana offers extensive customization options for these panels, allowing you to change visualization types, adjust settings, and set up alerts based on your metrics to notify you of any issues in real time. Finally, you can explore the dashboard, utilize template variables for dynamic visualizations, and even share your dashboards with team members to facilitate collaborative monitoring. Below Figure 2.14 shows us Grafana exposing NodeExporter metrics of localhost.



**Figure 2.14 Grafana Tool Exposing the NodeExporter Metrics Of LocalHost**

Below Figure 2.15 shows us Grafana service running on port 3000 on localhost



**Figure 2.15 Grafana Service Deployed On LocalHost**

# 3 INSTRUMENTING APPLICATION CODE FOR PROMETHEUS

Instrumenting application code for Prometheus involves integrating various libraries and techniques to expose metrics that can be scraped by the Prometheus server for monitoring and alerting purposes. To begin, you need to include the appropriate Prometheus client library in your application, which is available for multiple programming languages such as Go, Java, Python, and Node.js. Once the library is included, you can start defining custom metrics that capture critical information about your application's performance and behavior. Common metric types include counters, which increment for events like requests served; gauges, which represent values that can go up or down, such as memory usage; and histograms or summaries that provide insights into request latency and distribution. You’ll typically set up an HTTP endpoint (e.g., /metrics) in your application where Prometheus can scrape these metrics at regular intervals. Additionally, it’s essential to follow best practices, such as labeling metrics with relevant dimensions (like status codes or response times) to enhance query capabilities and provide more granular insights. As your application evolves, you may also want to instrument key business logic and background jobs, ensuring that all critical components of your system are monitored. Below figure 3.1 shows us the Prometheus GoLang library.



**Figure 3.1 Sample Prometheus GoLang Library (reference: google)**

# 4 FUTURE SCOPE OF THE PROJECT

Since Prometheus, AlertManager, and Grafana are not only used to observe metrics of infrastructure, the next level of implementation would be to also consider instrumenting application code and scrape application-specific metrics. This approach would significantly widen the scope of our monitoring capabilities, extending beyond basic infrastructure metrics to include detailed insights into application performance and behavior. By doing so, we can capture metrics such as request latency, error rates, and custom business metrics, which provide a deeper understanding of how our applications are performing in real-time.

Furthermore, broadening our infrastructure monitoring to include platforms like Kubernetes is crucial. Prometheus works natively with Kubernetes-deployed infrastructure, making it an ideal choice for observing the dynamic and complex environments that Kubernetes orchestrates. By leveraging Prometheus's native integration with Kubernetes, we can monitor cluster health, resource usage, and the status of deployed applications more effectively.

In addition, focusing on tier 0 (highly critical) applications and infrastructure will ensure that we prioritize the most mission-critical components of our system. This level of observability allows us to detect and respond to issues that could have the most significant impact on our business operations. By implementing comprehensive monitoring and alerting for these critical applications and infrastructure, we can ensure high availability, reliability, and performance.

# 

# 5 REFERENCES

[PuR2022] Carl Martin, Prometheus: Up & Running: Infrastructure and Application Performance Monitoring

[Himp2023] Pedro Araujo, Hands-On Infrastructure Monitoring with Prometheus: Implement and scale queries, dashboards, and alerting across machines and containers

[Pmlo 2022] Pedro Araujo, Practical MLOps: Operationalizing Machine Learning Models