



الجمهورية الجزائرية الديمقراطية الشعبية

People's Democratic Republic of Algeria

وزارة التعليم العالي والبحث العلمي

Ministry of Higher Education and Scientific Research

المدرسة الوطنية العليا للتكنولوجيا المتقدمة

Ecole Nationale Supérieure des Technologies Avancées



ELECTRICAL AND INDUSTRIAL COMPUTER ENGINEERING

Final Year Project to Obtain the Diploma of Engineering

- Field -

Telecommunication

- Specialty -

Telecommunication Systems and Networking

- Subject -

Enhanced IT Infrastructure Monitoring with Anomaly Detection: Combining Prometheus, Grafana and Machine Learning

Realized by

Aya Alia Abdi & Rima Boubeghel

Members of the Jury :

Mr Tarek Cherifi	Chair
Mr Abdelkader Balahcene	Examiner
Mme Lila Abbad	Examiner
Mr El Hadi Khoumeri	Supervisor

Algiers, Jun 25th 2024

Academic year 2023-2024

Dedication

Aya Alia Abdi

To my mother,

May this modest work be the fulfillment of the wishes you constantly expressed in your prayers. You are my wellspring of love, my source of energy, my flame of determination.

This work is the fruit of the sacrifices you made for my education and upbringing.

To my father,

I want to thank you for your unwavering support and kindness throughout my school journey. Your presence has been invaluable and has helped me move forward.

Thank you from the bottom of my heart.

To my dear sisters, Assia and Hasna.

Thank you for your constant encouragement and moral support.

To Rima, my partner and friend,

Who shared all the work with me and endured my moods during stressful times. Thank you for all the good times we've had together.

To my dear friends, Asma and Khaled.

Without you, this adventure would have been so different.

Rima Boubeghel

To my dearest mother, who always believes in me, supports me, and wishes me success.

To my dear father, who always asks about me, supports me, and does the impossible for me.

To all my sisters: Kheira, Siham, and Nihad, and to all my brothers: Mohammed and Fares, who always encourage and guide me to go beyond my dreams.

To all my friends, who encourage me, support me, and always try to make me happy, especially Aya and Asma . . .

Acknowledgement

We wish to express our gratitude to Allah the Almighty for granting us the courage and patience necessary to successfully complete this work.

We would first like to express our thanks and appreciation to our supervisor, Dr. KHOUMERI EL Hadi, Assistant Professor in the GEII department at ENST, for his patience, availability, and invaluable guidance.

We wish to extend our deep gratitude to the jury members for their valuable contribution, support, and time dedicated to evaluating our work.

We also thank the entire teaching team at the National Higher School of Advanced Technology and the professional speakers responsible for our education, who have contributed to the completion of this work.

ملخص

مشروعنا يهدف إلى تحسين المراقبة في الوقت الفعلي باستخدام تقنيات متقدمة. نقوم بتصميم وتطوير وتنفيذ نظام مراقبة يستخدم بروميثيوس لجمع البيانات وجرافانا للتصور. بالإضافة إلى ذلك، نقوم بدمج التعلم الآلي لاكتشاف الشذوذ في المقاييس. يسهل هذا النظام مراقبة البنية التحتية لتكنولوجيا المعلومات ويحدد الأنماط غير المعتادة تلقائياً.

الكلمات المفتاحية: المراقبة، البنية التحتية لتكنولوجيا المعلومات، بروميثيوس، جرافانا، اكتشاف الشذوذ.

Abstract

Our project aims to enhance real-time monitoring using advanced technologies. We design, develop, and implement a monitoring system that employs Prometheus for data collection and Grafana for visualization. Additionally, we integrate machine learning to detect anomalies in metrics. This system facilitates IT infrastructure monitoring and automatically identifies unusual patterns.

Keywords: Monitoring, IT Infrastructure, Prometheus, Grafana, Anomaly Detection.

Résumé

Notre projet vise à améliorer la surveillance en temps réel en utilisant des technologies avancées. Nous concevons, développons et implémentons un système de surveillance qui utilise Prometheus pour la collecte de données et Grafana pour la visualisation. De plus, nous intégrons l'apprentissage automatique pour détecter les anomalies dans les métriques. Ce système facilite la surveillance de l'infrastructure informatique et identifie automatiquement les schémas inhabituels.

Mots Clé : Surveillance, Infrastructure Informatique, Prometheus, Grafana, Détection d'Anomalies.

Contents

List of Figures	i
List of Tables	iv
Acronyms	v
General Introduction	1
0.1 State of the Art	1
0.2 Dissertation Organization	2
1 General Information on IT Infrastructure Monitoring	3
1.1 Introduction	3
1.2 IT Infrastructure	3
1.2.1 Definition	3
1.2.2 Components	4
1.3 IT Infrastructure Monitoring	5
1.3.1 Definition	5
1.3.2 Monitoring Data Types	6
1.3.3 Monitoring Data Storage	8
1.3.4 Monitoring Tools	8
1.4 Prometheus	10
1.4.1 Definition	10
1.4.2 Architecture	11
1.4.3 Data Collection	11
1.4.4 Recording and Alerting Rules	14
1.4.5 Alerting and Alertmanger	15
1.4.6 Service Discovery	17
1.4.7 Instrumentation	18
1.5 Grafana	18

1.6	Grafana and Prometheus Integration	19
1.7	Conclusion	20
2	Anomaly Detection in Time Series Data	21
2.1	Introduction	21
2.2	Definition	21
2.3	Anomaly Types	22
2.4	Nature of Input Data	24
2.5	Output of Anomaly Detection	24
2.6	Anomaly Detection Techniques	25
2.7	Anomaly Detection Process	29
2.8	Conclusion	30
3	Conception and Implementation	31
3.1	Introduction	31
3.2	Exploited Resources	31
3.2.1	Hardware Resources	32
3.2.2	Software Tools	32
3.3	Network Architecture	34
3.4	Implementation of The Monitoring System	35
3.4.1	Prometheus	37
3.4.2	Exporters	41
3.4.3	Web Application Instrumentation	46
3.4.4	Establishing Alerting Mechanism in Prometheus	54
3.4.5	Data Visualisation Using Grafana	57
3.4.6	Deployment of The Monitoring System using Docker Compose	61
3.5	Implementation of Anomaly Detection in Time Series Data Using LSTM Autoencoders	63
3.6	Conclusion	69
4	Results and Validation	70
4.1	Introduction	70
4.2	Validation of the Monitoring system	70
4.2.1	Prometheus	70
4.2.2	Container Status with CAdvisor	73
4.2.3	Exporters	73
4.2.4	Web Application	77
4.2.5	Alerting	78

4.3 Grafana Dashboards	80
4.4 Evaluating of the Anomaly Detection Model	83
4.5 Conclusion	86
4.6 Future Work	86
General Conclusion	88
Bibliography	90
A Configuration Files	A

List of Figures

1.1	IT Infrastructure Components	4
1.2	Monitoring Process Phases	5
1.3	Monolith vs Distributed Monitoring Systems	9
1.4	Push vs Pull Mechanism in Monitoring Systems	10
1.5	Prometheus logo	10
1.6	Prometheus Architecture	12
1.7	SNMP Monitoring Architecture	13
1.8	SNMP MIB Tree	14
1.9	Life Cycle of Recording Rules	15
1.10	Alerting Workflow in Prometheus	16
1.11	Example of Alert Routing Structure in Alertmanager	16
1.12	Grafana logo	18
1.13	Prometheus and Grafana Integration	19
2.1	Illustration of simple anomalies in 2D	22
2.2	Anomaly Types [1]	22
2.3	Normal Data Distribution	26
2.4	Skewed Data Distribution	26
2.5	The Influence of Data Distribution on the Z-score Method	26
2.6	Performance Comparison of Deep learning-based algorithms Vs Traditional Algorithms [1]	27
2.7	An illustration of a LSTM Autoencoder network [2]	29
2.8	Anomaly Detection Process	30
3.1	GNS3 Logo	32
3.2	Initial Setup Screen of GNS3 VM	33
3.3	Virtualbox Logo	33
3.4	Docker Logo	34
3.5	CAdvisor Logo	34

3.6	Network Architecture	35
3.7	Network Configuration of the Router	36
3.9	Conception and implementation of the monitoring system	36
3.8	Network Configuration of the Switch	37
3.10	Prometheus Configuration File 1	38
3.11	Prometheus Configuration File 2	39
3.12	Checking Prometheus.yml file using Promtool	40
3.13	Recording Rules file rules.yml	40
3.14	File-based Service Discovery	41
3.15	Content of Systemd Service File for Node Exporter	42
3.16	Node Exporter Service Running	42
3.17	WMI Node Exporter Service Running	43
3.18	SNMP v3 Configuration on The Router	44
3.19	SNMP v3 Configuration on The Switch	45
3.20	Tree Structure of the Uploaded Django Web Application	47
3.21	The Home Page of the Web Application	48
3.22	Diagram of Alerting Rules Implementation	55
3.23	Diagram of Route Tree Implementation	56
3.24	Configuring Prometheus as a Data Source for Grafana	58
3.25	Adding Prometheus as a Data Source to a Dashboard	58
3.26	Creating an Instance Variable in Dashboard	59
3.27	Configuring the Metric Panel	59
3.28	Incoming Traffic Visualization of the Router	60
3.29	Incoming Traffic Visualization of the Switch	60
3.30	Incoming Traffic Visualization of both the Router and the Switch	60
3.31	Dashboard Creation for Each Target	61
3.32	Docker Compose Services	62
3.33	Explanatory Diagram of the Steps Carried Out	65
3.34	CPU Usage Dataset	66
3.35	Evaluation of Training Process	68
4.1	Scrape Targets Listed on Prometheus Targets Page	71
4.2	Continuation 1 of Scraping Targets Listed on the Prometheus Targets Page	71
4.3	Continuation 2 of Scraping Targets Listed on the Prometheus Targets Page	72
4.4	Validation of Django App Recording Rules on the Prometheus Rules Page	72
4.5	Validation of Linux Recording Rules on the Prometheus Rules Page	72
4.6	Validation of Windows Recording Rules on the Prometheus Rules Page	73

4.7	Monitoring Containers using Cadvisor	73
4.8	Monitoring the Web App Container using Cadvisor	74
4.9	Network Throughput and Errors Monitoring with cAdvisor for the Web App	74
4.10	VM-Kali Metrics Exposed by Node Exporter	74
4.11	VM-Windows Metrics Exposed by WMI Exporter	75
4.12	SNMP Exporter User Interface	75
4.13	Router Metrics in SNMP Exporter HTTP Endpoint	76
4.14	Web Application Metrics Exposed on Metrics Endpoint	77
4.15	Web Application Metrics scraped by Prometheus	78
4.16	Different Alerts Displayed in Prometheus Alerts Interface	78
4.17	Fired Alerts Displayed in Alertmanager Interface	79
4.18	Gmail Notification for the Fired Alerts	80
4.19	Data Visualisation for VM-Kali	80
4.20	Windows VM Data Visualization Dashboard	81
4.21	Data Visualisation for Router	81
4.22	Data Visualisation for Switch	82
4.23	Data Visualisation for Docker	82
4.24	Data Visualisation for Web Application	83
4.25	Detected Anomalies in Testing Dataset Using Reconstruction Error Threshold	84
4.26	Timestamps of the detected Anomalies	84
4.27	Occurrences of Detected Anomalies in the Testing Dataset	85

List of Tables

3.1	Server Specifications	35
3.2	Parameters in an LSTM model	67
3.3	Optimal parameter values for the LSTM model	67
3.4	Training parameters for the LSTM autoencoder	67
3.5	Optimal training parameters for the LSTM autoencoder	68
4.1	Evaluation Metrics for Anomaly Detection	85

Acronyms

AI Artificial Intelligence

API Application Programming Interface

CPU Central Processing Unit

GNS3 Graphical Network Simulator-3

GNS3 VM Graphical Network Simulator-3 Virtual Machine

HDD Hard Disk Drive

HTTP Hypertext Transfer Protocol

IoT Internet of Things

IP Internet Protocol

IT Information Technology

JSON JavaScript Object Notation

LSTM Long Short-Term Memory

MAD Median Absolute Deviation

MAE Mean Absolute Error

MIB Management Information Base

MSE Mean Squared Error

NAT Network Address Translation

NSSM Non-Sucking Service Manager

OID Object Identifier

OSS Open Source Software

PromQL Prometheus Query Language

RAM Random Access Memory

RDBMS Relational Database Management System

RNN Recurrent Neural Network

SaaS Software as a Service

SMS Short Message Service

SMTP Simple Mail Transfer Protocol

SNMP Simple Network Management Protocol

SQL Structured Query Language

SSD Solid State Drive

TSDB Time Series Database

UI User Interface

VM Virtual Machine

WMI Windows Management Instrumentation

YAML Yet Another Markup Language

General Introduction

In today's interconnected world, ensuring the stability and security of IT infrastructure is crucial for organizations across various sectors. Effective monitoring systems are essential to maintain optimal performance and protect these networks. Our project aims to meet this requirement by leveraging advanced technologies to improve real-time monitoring capabilities.

The main goal of this project is to design, develop, and implement an advanced monitoring system tailored for IT infrastructure. This system integrates Prometheus and Grafana, widely recognized tools for collecting metrics and visualizing data, respectively. Additionally, it incorporates machine learning techniques to detect anomalies in IT infrastructure metrics.

Key components of the project include deploying Prometheus to efficiently gather data from IT infrastructure components, configuring monitoring targets, and establishing customized alerting rules. Furthermore, integrating Grafana allows for clear visualization and detailed analysis of network performance through customized dashboards. The project also involves training a machine learning model specifically designed to detect anomalies in time series data collected by Prometheus. This enhancement enables the monitoring system to automatically identify unusual patterns or behaviors in network metrics, which could signify potential issues in real time.

0.1 State of the Art

Recently, several companies adopted Prometheus and Grafana into their DevOps pipelines, such as Docker [3], Ericsson [4], SoundCloud [5], and GrafanaLabs [6]. Many studies leveraged Prometheus and Grafana to address a variety of challenges including the monitoring and detection of anomalies while reducing human intervention [7], optimizing resource allocation when monitoring big-data microservices-based applications [8], [9], and scaling monitoring frameworks to accommodate the complexity of High Performance Computing centers [10].

Also, recent research and development in IT infrastructure monitoring have focused

on enhancing the capabilities of Prometheus and Grafana through advanced machine learning techniques. Studies have demonstrated the effectiveness of LSTM autoencoders in various domains, including network traffic analysis, system performance monitoring, and predictive maintenance. For instance, a study by Zhao et al. (2019) [11] showcased the application of LSTM networks for real-time anomaly detection in cloud environments, highlighting their ability to reduce false alarms and improve detection accuracy. Another research by Malhotra et al. (2016) [12] explored the use of LSTM-based predictive models for identifying performance bottlenecks in microservices architectures, demonstrating significant improvements over traditional methods.

0.2 Dissertation Organization

This project is structured into four chapters as follows:

- **Chapter 1:** "General Information on IT Infrastructure Monitoring" Provides an overview of IT infrastructure monitoring, focusing on Prometheus. It details Prometheus' architecture, data collection processes, recording and alerting rules, and integration with Alertmanager and service discovery. Additionally, it discusses Grafana's role in visualizing data collected by Prometheus.
- **Chapter 2:** "Anomaly Detection in Time Series Data" Focuses on anomaly detection within time series data. It defines anomalies, categorizes different types of anomalies, and examines the nature of input data essential for effective anomaly detection. The chapter elaborates on the expected outputs of anomaly detection and reviews various techniques employed in the anomaly detection process.
- **Chapter 3:** "Conception and Implementation" Details the practical deployment of a comprehensive monitoring system for IT infrastructure. It describes the network architecture designed to support the monitoring system and the deployment of the monitoring system using Docker Compose. The chapter also discusses anomaly detection using LSTM autoencoders, explaining the method and steps involved.
- **Chapter 4:** "Results and Validation" Provides a comprehensive overview of the outcomes and validation processes of the implemented monitoring system. It also discusses future work, focusing on securing the monitoring system and integrating the anomaly detection model in real-time.

Chapter 1

General Information on IT Infrastructure Monitoring

1.1 Introduction

In this chapter, we introduce the basics of IT infrastructure monitoring, covering its key concepts and the importance of monitoring both hardware and software components. Furthermore, we introduce Prometheus, an open-source monitoring and alerting toolkit, outlining its architecture, data collection mechanisms, alerting capabilities, and its integration with Grafana for enhanced visualization.

1.2 IT Infrastructure

1.2.1 Definition

Information Technology (IT) infrastructure refers to all the necessary components required to support and manage enterprise IT services and environments. A strong IT infrastructure plays a crucial role in enhancing employee productivity and efficiency, while also enabling the delivery of high-quality solutions to customers. It is imperative for businesses to establish a well-designed IT infrastructure to lower the risks associated of security issues, and ensuring optimal productivity.

In IT infrastructure, three primary types exist: traditional, cloud, and hybrid infrastructures [13].

- In traditional IT infrastructure, the business or organization owns and manages all components on-site. This setup requires a significant amount of hardware, substan-

tial physical space, and considerable power. Consequently, traditional IT infrastructure tends to be more expensive compared to cloud infrastructure.

- Cloud Infrastructure enables businesses to access IT resources over the internet by renting them from a cloud service provider. This setup is known as a public cloud, but businesses also have the option to create a private cloud. Cloud infrastructure offers flexibility, allowing access to services from various locations.
- Hybrid Infrastructure integrates traditional and cloud infrastructure, enabling companies to utilize the cloud’s scalability and flexibility while maintaining sensitive operations on-premises [13], for example: sensitive data and critical applications can stay on-premises for better security, while less sensitive or more scalable applications can be hosted in the cloud.

1.2.2 Components

IT infrastructure consists of several key components that work together to support an organization’s technology needs. These components include hardware, software, networks, data centers, and cloud services as shown in Figure. 1.1. Understanding these components is essential for designing and implementing a comprehensive IT infrastructure strategy.

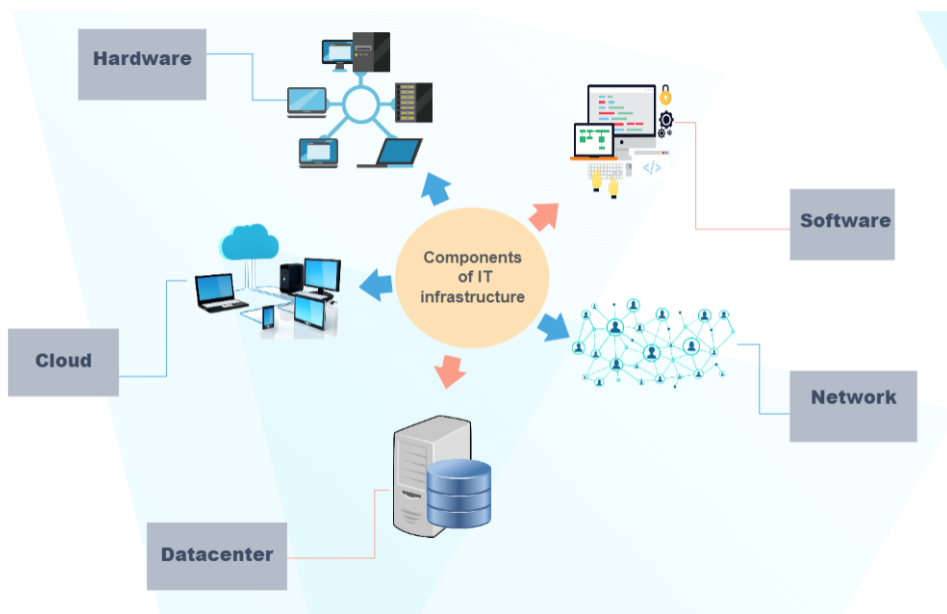


Figure 1.1: IT Infrastructure Components

- Hardware encompasses the physical elements like servers, computers, printers, storage devices, and networking equipment, which are essential for data input, storage,

sharing, and output. Regular evaluation and updates of hardware are crucial to keep pace with technological advancements.

- Software, composed of code-based instructions, enables hardware to function, including operating systems, content management systems, and web servers.
- Network manages the communication and operations between internal and external systems through hardware like routers, switches, and hubs, as well as features such as internet connectivity, firewalls, and security measures.
- Data centers, housing numerous servers and related components, provide a centralized location for managing IT equipment and data.
- Cloud services, offered by companies like Amazon Web Services, Microsoft Azure, and Google Cloud, complement data centers by offering scalable, virtualized online resources, providing organizations with greater flexibility and scalability.

1.3 IT Infrastructure Monitoring

1.3.1 Definition

Monitoring refers to the comprehensive process of collecting, storing, and analyzing data related to software and hardware health, functioning and performance to identify issues and improve system behavior [14].

The monitoring process includes several key phases as shown in Figure. 2.8.

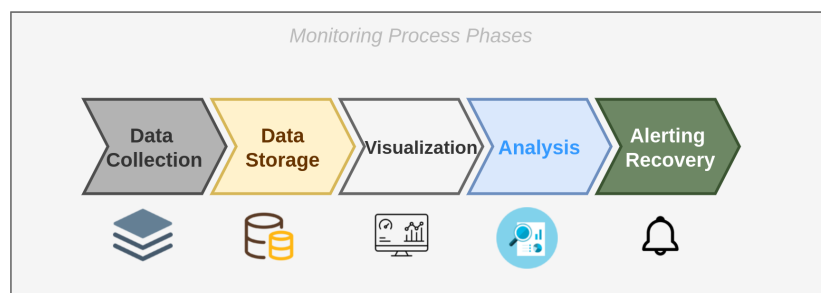


Figure 1.2: Monitoring Process Phases

- Data Collection: Capturing data from various sources.

- **Data Storage:** Storing the collected data in a structured manner for easy retrieval and analysis.
- **Visualization:** Creating dashboards and visual representations of the data to provide insights and facilitate monitoring.
- **Analysis:** Examining the data to detect anomalies, identify trends, and diagnose problems.
- **Alerting and Recovery:** Setting up alerts for specific conditions and initiating recovery actions to address detected issues.

1.3.2 Monitoring Data Types

Monitoring tools can collect a variety of different types of data. Each type of data serves a unique purpose and provides different insights into the system's performance and behavior. That data primarily takes four forms in IT systems: events, logs, traces and metrics. Let's take a look at each one of them.

- **Events**

Events are notifications about changes or specific occurrences within the environment [15]. They are infrequent but provide crucial context for understanding changes in a system's behavior. Here are some examples of events:

- **Changes:** Code releases, builds, and build failures.
- **Alerts:** Notifications generated by the primary monitoring system or integrated third-party tools.
- **Scaling Events:** Adding or removing hosts or containers.

- **Logs**

Logs are specified information about system runtime variable values and errors, they provide information about an event that occurred at a specific time, containing a timestamp indicating when the event occurred and some payload [16]. Most frameworks and libraries support logging, and it can be as straightforward as outputting a line of text. Logs generally come in three forms:

- **Plain text:** The most common type, typically emitted by running processes.
- **Structured:** Logs in structured formats like JavaScript Object Notation (JSON), which facilitate further processing.

- Unstructured: Logs stored in binary format, such as those generated by some Relational Database Management System (RDBMS) like MySQL. These are usually only useful to the system that generates them.

Logs are valuable because they provide contextual information about specific requests handled by a service. Depending on the granularity of your logs, you can identify the root cause of performance issues or debug problems without using a debugger. While it might seem beneficial to log every single action your services perform, you need to consider the performance implications. [17].

- **Traces**

Traces represent the communication flow between services and dependencies in a distributed system, known as distributed tracing. This method captures trace information across various processes, nodes, networks, and security boundaries. Properly implemented traces provide visibility into which services are involved from the beginning to the end of a request, as well as the duration of each service's operations.

Distributed tracing is a method used to monitor and troubleshoot applications built using microservices or other distributed architectures. It involves tracking requests as they propagate through various services and components of a distributed system, providing a comprehensive view of the system's behavior. A key aspect of distributed tracing is the use of Trace IDs, which are unique identifiers attached to each request that allow tracking across different services. Spans, representing individual units of work within a trace, capture operations or processes involved in handling a request. Visualization tools are essential in distributed tracing, as they illustrate the path of a request and the time spent in each component, enabling easier identification of bottlenecks and errors [17].

- **Metrics**

Metrics are quantitative measures of properties of software or hardware components that are critical for monitoring and understanding system performance [18]. Since metric is a numeric representation, it is very convenient for any type of aggregation, summarization, and correlation [17]. Metrics are typically collected as observations over time, where each observation includes a value, a timestamp, and potentially additional properties such as source or tags. This collection of observations forms a time series, which can be visualized to provide insights into system behavior.

A classic example of time series data is the collection of website visits, or hits. Observations about website hits are periodically collected, recording the number of

hits along with the timestamps of these observations. Additional properties such as the source of the hit, the specific server that was hit, and other relevant information may also be collected.

Observations are typically collected at fixed time intervals, referred to as granularity or resolution, which can vary from one second to five minutes or even 60 minutes or more [15]. Selecting the appropriate granularity is crucial, too coarse a granularity might miss important details. For instance, sampling Central Processing Unit (CPU) or memory usage every five minutes may fail to capture anomalies. Conversely, a finer granularity can lead to the need for storing and analyzing large volumes of data. Metrics provide an overview of your system's behavior, allowing you to identify anomalies such as operational failures, unusual CPU or memory usage, or performance degradation [19].

1.3.3 Monitoring Data Storage

Most monitoring data is time series data, characterized by two components: the timestamp indicating when the data was collected and the data itself, which can be in text format (log data) or numerical format (metrics).

To fully leverage monitoring data, it must be stored in a database capable of efficiently handling time series data. Specifically designed for this task, a database model known as a time series database Time Series Database (TSDB) is employed. TSDBs have garnered significant attention in the database domain, experiencing the most rapid growth in popularity over the last two years. This surge in popularity can be attributed to the increasing need for monitoring large-scale systems that generate substantial volumes of time series data, as well as the proliferation of Internet of Things (IoT) devices, which also contribute to the creation of time series data.

A significant distinction between TSDB and conventional Structured Query Language (SQL) or NoSQL databases lies in their approach to data storage. Unlike traditional databases, TSDB consistently saves new information as INSERTs rather than UPDATEs. This fundamental difference enables TSDB to maintain a comprehensive history of saved items over time.

1.3.4 Monitoring Tools

Monitoring tools are essential for maintaining the health and performance of IT systems, and they can be categorized based on various criteria. Here are the main criteria used to categorize these tools:

- **Monitoring Systems Architecture**

Monitoring tools can be categorized into two types based on their architecture. Figure. 1.3 distinguishes between Monolith and Distributed Monitoring Systems.

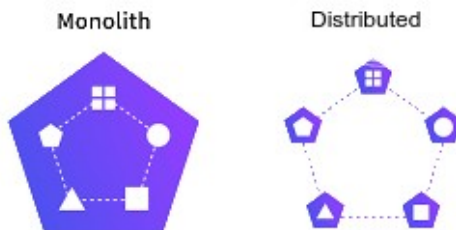


Figure 1.3: Monolith vs Distributed Monitoring Systems

In a monolithic monitoring system, all monitoring functionalities are contained within a single service or application. This means that data collection, data storage, visualization, analytics and alerting, are all handled by one integrated system. Example, Nagios: Nagios [20] is a well-known monolithic monitoring tool. It performs all aspects of monitoring from a single application. It collects data from various endpoints, processes it to check for any anomalies, stores the data for historical analysis, and provides alerting and visualization features.

In a distributed monitoring system, different functionalities are split across multiple specialized services. Each service handles a specific aspect of monitoring, and they work together to provide a complete monitoring solution. Recently, the trend has shifted towards distributed monitoring systems, Smaller units are easier to scale, build, and maintain. Different technologies can be utilized for different units. The overall system becomes more resilient to errors since the failure of one part does not affect the others. Additionally, changing parts of the system is simpler with a distributed architecture. Example, Prometheus.

- **Data Collection Methods**

In monitoring, there are two primary methods for obtaining information from a client system: push-based and pull-based monitoring. In push-based monitoring, the client system takes the initiative by sending monitoring information to the monitoring system. Conversely, in pull-based monitoring, the monitoring system initiates the process by requesting information from the target system [21]. Essentially, the distinction lies in which party makes the first move to exchange data as shown in Figure. 1.4.

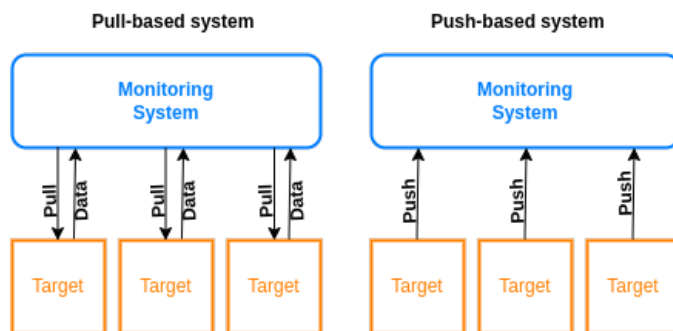


Figure 1.4: Push vs Pull Mechanism in Monitoring Systems

- **Development Stage**

Monitoring tools can be categorized into three types based on the development stages of their implementation within a company: Software as a Service (SaaS) monitoring, Open Source Software (OSS) monitoring, and custom-built monitoring. In the SaaS monitoring stage, companies utilize ready-made SaaS tools that are easy to operate and integrate into existing systems, offering convenience and quick deployment but limited customization. As companies advance to the OSS monitoring stage, they adopt open-source software tools, which provide greater customization and flexibility but require more configuration and setup time. In the custom-built monitoring stage, companies develop their own monitoring solutions tailored to their specific needs. This approach is typically adopted by large technology companies that encounter scalability or customization challenges with existing solutions. Prometheus is an example of a custom-built monitoring tool that originated within SoundCloud and was later released as an open-source project, illustrating the trend of internal tools evolving into widely adopted OSS tools.

1.4 Prometheus

1.4.1 Definition

Prometheus is an open-source monitoring and alerting toolkit designed for collecting time-series data. Created by SoundCloud in 2012, the project has grown to benefit from a large and active community of developers and users. Now maintained independently by various companies, Prometheus joined the Cloud Native Computing Foundation in 2016 [22].to enhance its development and clarity, becoming the foundation’s second hosted project after Kubernetes [23].



Figure 1.5: Prometheus logo

Prometheus operates by scraping metrics via HTTP, extracting time-series data from various targets, which can include diverse sources such as services, operating systems and applications. This time-series is a sequence of data points that typically include continuous measurements over a specific time period. In Prometheus, time series data is identified by a metric name and a series of key-value pairs (also called labels). The database uses flexible query language called Prometheus Query Language (PromQL) to store all incoming data in an efficient, compressed database on disk. Prometheus is a flexible collection tool that can work with different components to complete the setup and display the results.

1.4.2 Architecture

Figure. 1.6 illustrates the architecture of the Prometheus monitoring system and its ecosystem components [24]. Prometheus pulls metrics from various jobs and exporters, where jobs are a collection of targets with the same purpose. storing the data in a Time-Series Database (TSDB) for efficient querying. Service discovery mechanisms, such as Kubernetes and file-based discovery, help identify targets. The Pushgateway [25] facilitates the collection of metrics from short-lived jobs by allowing them to push data at completion. The Prometheus server includes a retrieval component, the TSDB, and an Hypertext Transfer Protocol Hypertext Transfer Protocol (HTTP) server for accessing the data. Alertmanager receives alerts from Prometheus and sends notifications to endpoints like PagerDuty and email. For visualization, the Prometheus Web User Interface (UI) and Grafana provide powerful interfaces, with Grafana offering advanced dashboard capabilities. The system stores metrics on Hard Disk Drive (HDD)/Solid State Drive (SSD) for persistence, and application programming interface Application Programming Interface (API) clients can query the Prometheus API for data integration with other tools.

1.4.3 Data Collection

Prometheus utilizes a pull mechanism to collect data from its targets, although a push mechanism can be employed when necessary. The data collected is in the form of time series. Prometheus can be configured to collect data from various devices and applications by installing exporters or using client libraries. Some exporters, like the node exporter and the Windows Management Instrumentation (WMI) exporter, must be installed on the target machine to collect metrics. Others, such as SNMP, only require activation on the target machine without additional software installation. Configuration of targets is

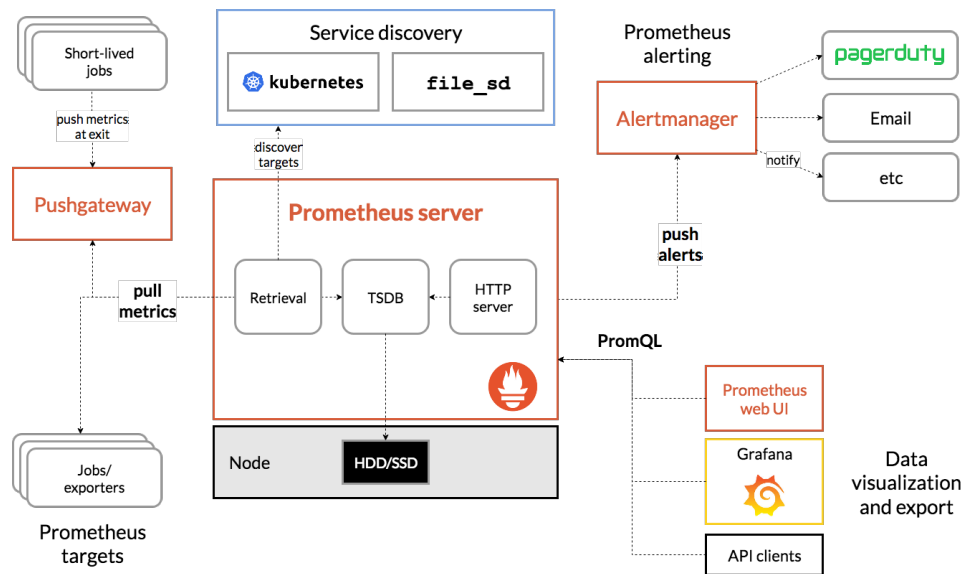


Figure 1.6: Prometheus Architecture

done in Prometheus' `prometheus.yml` file, and sometimes in the Yet Another Markup Language (YAML) files specific to the exporters [26].

An exporter is a software tool that can be deployed on an application or device where monitoring is needed. Prometheus sends requests to the exporter, which collects the requested data from the target, converts it into a format that Prometheus can read, and sends the data back to the Prometheus server. Prometheus supports numerous exporters, such as Node Exporter, WMI Exporter and SNMP Exporter.

- **Node Exporter**

Node Exporter is designed to collect data from Unix environments. It provides hardware and kernel metrics from the target machine, such as CPU, memory, and disk space metrics. Node Exporter is designed specifically for monitoring the machine itself, rather than individual processes or services. It can be downloaded from the Prometheus website and should be installed on the target machine. By default, Node Exporter runs on port 9100. [27].

- **WMI Exporter**

WMI Exporter provides metrics for Windows operating systems through various collectors that gather different system metrics. It can be run as a standalone application or as a Windows service. By default, WMI Exporter exposes metrics on port 9182. [28].

- **SNMP Exporter**

The Simple Network Management Protocol (SNMP) Exporter is particularly useful for monitoring network devices such as switches, routers, and firewalls. SNMP must be enabled on the target device before the exporter can collect metrics from it.

The diagram in Figure. 1.7 illustrates the process of network monitoring using Prometheus and its SNMP exporter. On the left side, a network device is being polled for SNMP data. This data is then collected and sent to the Prometheus SNMP exporter, which is responsible for converting the SNMP data into a format that Prometheus can understand. The Prometheus SNMP exporter runs within a containerized environment using Kubernetes or Docker. Once the SNMP data is processed, it is scraped by Prometheus, which then stores the data for further analysis and monitoring.

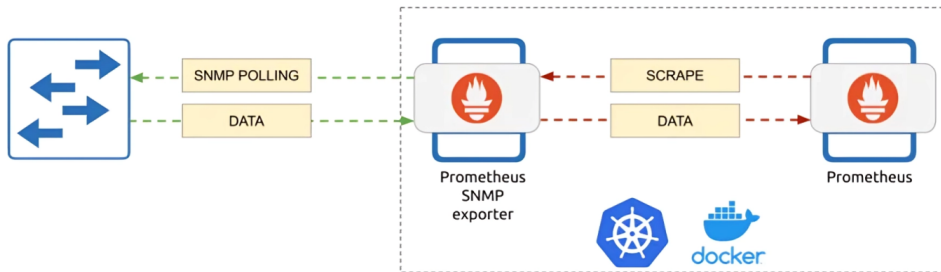


Figure 1.7: SNMP Monitoring Architecture

Using the SNMP Exporter requires understanding Management Information Base (MIB)s and Object Identifier (OID)s. MIBs and OIDs specify the information that can be retrieved from devices. An OID defines specific metrics within a MIB tree. There are universal MIBs supported by most network devices, but some devices require manufacturer-specific MIBs for certain information. OIDs are entered into the `snmp.yml` file, where the exporter uses them to find and translate the data for Prometheus [26].

To consider an example, The Object Identifier (OID) 1.3.6.1.2.1.1 represents the System group within the Management Information Base (MIB) tree as shown in Figure. 1.8. This group includes objects that provide essential information for managing a network device such as system uptime, system name, and system description.

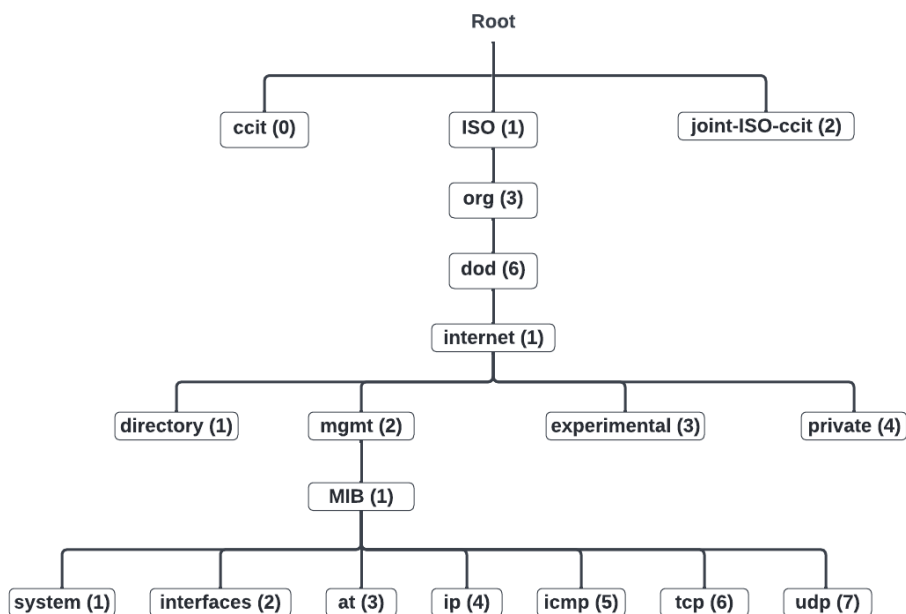


Figure 1.8: SNMP MIB Tree

1.4.4 Recording and Alerting Rules

Prometheus supports two types of rules that can be configured and evaluated at regular intervals: recording rules and alerting rules. To add these rules, create a YAML file with the appropriate rule statements and specify this file in the `rule_files` field of the Prometheus configuration.

- **Recording Rules**

Recording rules in Prometheus are directives defined within the Prometheus configuration file that instruct the Prometheus server to compute and generate new time series data based on existing metrics. These rules typically involve applying functions or operators to existing metric data to derive new insights or aggregations. They enable users to create custom metrics tailored to their specific monitoring requirements [29].

Recording rules are advantageous because they accelerate query performance. Instead of recalculating the original expressions every time, querying precomputed results is much faster. This is particularly useful for dashboards that frequently refresh and need to retrieve the same data repeatedly. It ensures quicker data access and enhances overall system efficiency.

Figure. 3.13 illustrates the process of recording rules, starting with the creation of a separate `rules.yml` file. This file defines PromQL expressions that are to be evaluated regularly, each expression accompanied by a distinct name. Once the file is prepared, it is fed into Prometheus. Subsequently, Prometheus evaluates these expressions at the specified scrape interval and stores the resulting data as new time series with unique metric names in the storage.



Figure 1.9: Life Cycle of Recording Rules

- **Alerting Rules**

Alerting rules enable the specification of conditions under which an alert should be fired. These conditions are based on expressions written in PromQL, similar to how recording rules are configured. Any results from these expressions become alerts.

1.4.5 Alerting and Alertmanager

Alerting is the process of notifying users when specific predefined criteria are met. In Prometheus, alerting is generally divided into two main components: alert rules and Alertmanager.

Prometheus uses alerting rules, defined in its configuration, to specify the conditions under which alerts should be triggered. When a metric meets these conditions, Prometheus triggers an alert and sends it to Alertmanager [18]. Alertmanager handles these alerts by routing them to appropriate notification channels such as email, Telegram, or Slack [30], as illustrated in Figure. 1.10.

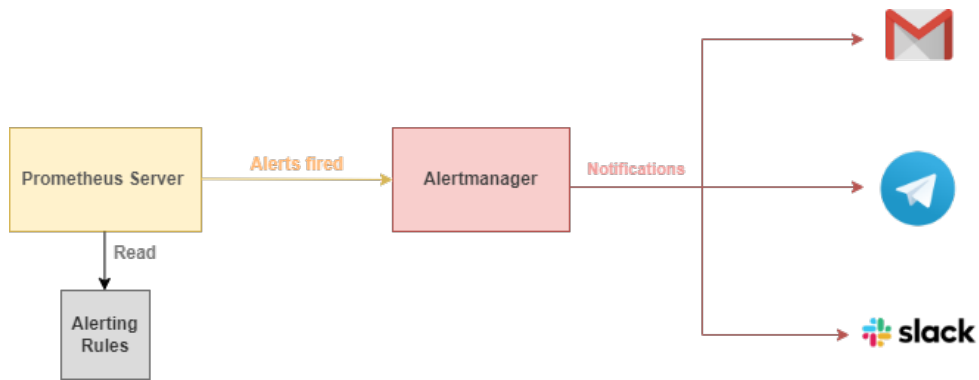


Figure 1.10: Alerting Workflow in Prometheus

Alertmanager is the open-source standard for handling alerts sent by Prometheus server and turns them into notifications. It manages the alerts by grouping similar ones, silencing them during maintenance, routing them to the appropriate receiver and so on [31]. We can define each alert processing as follow:

- **Routing** Alertmanager uses a hierarchical routing tree to manage alerts, as illustrated in Figure. 1.11. When an alert is generated, it first reaches the root route, which is the initial point for processing incoming alerts. The root route evaluates the alert against defined conditions and directs it to the appropriate child route based on matching criteria. Each child route further processes and forwards the alert until it reaches the intended child route, which has its receiver (e.g., an email address) to handle the notification. If an alert does not match any specific routing criteria, it is sent to the fallback receiver configured in the root route, ensuring that all alerts are appropriately managed [32].

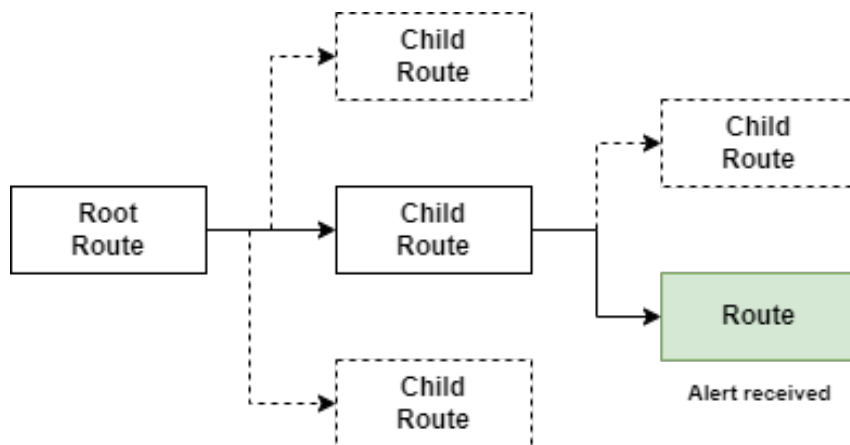


Figure 1.11: Example of Alert Routing Structure in Alertmanager

- **Grouping** Grouping in Alertmanager allows multiple alerts sharing similar la-

bels to be consolidated into a single group, enabling one notification per group. This is configured using the `group_by` field in Alertmanager's configuration file `alertmanager.yml`. Grouping helps reduce notification overload, simplifying the management of a large number of alerts for the recipient.

- **Inhibition** Inhibitions are configured through Alertmanager's configuration file. This process involves muting downstream alerts based on their label sets. For example, if there are two alerts for the same cluster (one with a critical severity label and another with a warning severity label) the system can be configured to mute the warning severity alert if the critical severity alert is triggered. This helps avoid notifications for alerts that are not relevant to the current issue.

- **Silencing**

Silencing provides the option to temporarily mute an alert when it is expected to be triggered during scheduled procedures, such as database maintenance, or when the alert has already been acknowledged during an incident and further notifications are unnecessary while resolving the issue. Silences are configured through the web interface of Alertmanager.

- **Throttling**

Throttles are configured in Alertmanager's configuration file. They allow customization of renotification settings through three key parameters:

- **group_wait**: Specifies the initial waiting period before sending a notification for a group of alerts, with a default duration of 30 seconds [33].
- **group_interval**: Determines the interval before sending notifications for new alerts added to an already notified group, with a default duration of 5 minutes [33].
- **repeat_interval**: Sets the interval before resending notifications for an alert that has already been notified, with a default duration of 4 hours [33].

1.4.6 Service Discovery

Service discovery is a mechanism that allows automatic discovery and monitoring of targets and services. In a static configuration, without using service discovery, each target's Internet Protocol (IP) address and port must be manually listed in the scrape configuration. While this approach works for a few hosts, it becomes impractical for a larger fleet,

particularly for dynamic environments using containers and cloud-based instances, where instances can frequently change, appear, and disappear [32].

Prometheus offers various service discovery options for identifying scrape targets, including Kubernetes, Consul, and many others. For service discovery systems not currently supported, the file-based service discovery mechanism in Prometheus may be the best solution. This mechanism allows scrape targets to be listed in a JSON or YAML file, along with metadata about those targets. Scraping targets can be configured manually, or a separate script or process can be used to edit the service discovery file.

1.4.7 Instrumentation

Monitoring applications is achieved through instrumentation, which involves integrating code within an application to enable it to report on its internal state. This equips the application with the capability to provide valuable insights, allowing Prometheus to gather detailed information about the application's functionality and performance [34].

Instrumentation is added to application code using one of the Prometheus client libraries. The appropriate client library is selected based on the language in which the application is written. This exposes the collected data via an HTTP endpoint, enabling Prometheus to scrape the metrics at regular intervals [32].

Prometheus client libraries provide four core types of metrics: Counter, Gauge, Histogram, and Summary. A Counter is a cumulative metric representing a single, monotonically increasing value that can only increase over time or be reset to zero upon a restart. In contrast, a Gauge represents a single numerical value that can arbitrarily increase or decrease. These libraries handle the details of formatting and exposing metrics in the Prometheus exposition format [18].

1.5 Grafana

Grafana, developed by Torkel Ödegaard in 2014 [35], is a powerful and free tool designed for effective and informative data visualization. While Grafana itself does not collect data, it seamlessly connects to a wide range of data sources and utilizes its specialized query editor to interact with them.

Operating as a service on a computer or server, Grafana is accessed via a web browser, typically on port 3000. This accessibility enables users to create interactive dashboards with versatile visualization options, drawing from diverse data sources. Grafana supports multiple



Figure 1.12:
Grafana logo

data sources, customizable alert systems, and annotations. Its flexibility extends to integration with numerous plugins, allowing it to adapt easily to different environments [36].

Moreover, Grafana serves as a robust alert system, capable of sending notifications through email, Short Message Service (SMS) and more, enhancing its utility as a comprehensive monitoring and visualization platform.

1.6 Grafana and Prometheus Integration

Integrating Grafana with Prometheus provides numerous advantages for monitoring and visualization. Grafana uses Prometheus as a data source to facilitate real-time monitoring of systems. It offers rich visualization capabilities, allowing the creation of customized dashboards with graphs, charts, and tables to analyze system health and performance. Prometheus’s powerful alerting system can be integrated with Grafana to set up alerts based on specific thresholds or conditions, delivering notifications via different channels. Both tools are highly scalable and flexible, capable of handling large data volumes, which makes them ideal for complex and distributed architectures.

Figure. 1.13 summarizes the integration of Prometheus and Grafana, starting with collecting metrics using Prometheus and creating alerts with Alertmanager, and culminating in visualizing the data with Grafana.

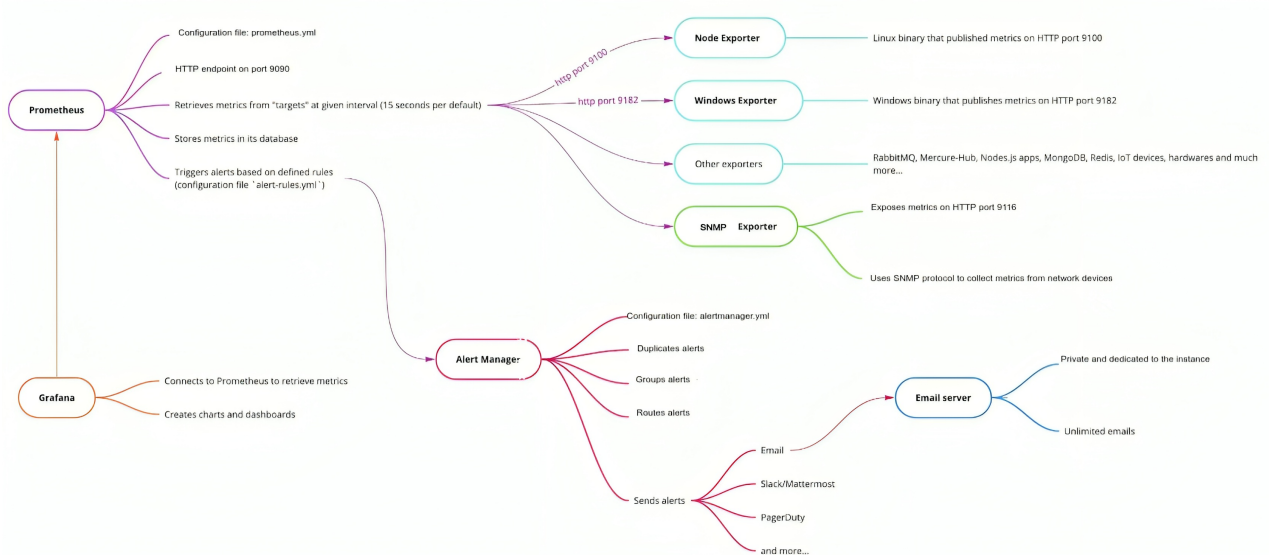


Figure 1.13: Prometheus and Grafana Integration

1.7 Conclusion

In conclusion, this chapter has provided a foundational understanding of IT infrastructure monitoring, emphasizing the critical role it plays in maintaining the health and performance of both hardware and software components. We have explored the different types of IT infrastructure and highlighted the essential components involved. Additionally, we have introduced Prometheus, detailing its architecture, data collection methods, and alerting capabilities, along with its integration with Grafana for improved visualization.

Chapter 2

Anomaly Detection in Time Series Data

2.1 Introduction

In this chapter, we explore the critical concept of anomaly detection within time series data. Anomalies, which are data points that significantly deviate from expected patterns, can indicate important and sometimes urgent insights in IT infrastructure monitoring. Understanding these irregularities not only helps in identifying errors and system failures but also uncovers significant events that could impact decision-making processes.

We define what anomalies are, discuss their types, and explain the methodologies employed to detect them effectively.

2.2 Definition

An anomaly is a pattern or data point that deviates significantly from the expected behavior or the majority of data. It stands out as unusual or irregular compared to the established norms within the dataset [37]. For example, as illustrated in Figure. 2.1, in a dataset with two normal regions, N_1 and N_2 , points that fall far outside these regions, such as o_1 , o_2 , and the points in region o_3 , are considered anomalies. These anomalies can arise due to various reasons, such as errors, system breakdowns, or other atypical activities, and they are particularly noteworthy for analysis due to their significant divergence from the norm.

Anomaly detection is the process of identifying these irregular patterns or data points that do not conform to the expected behavior in a dataset. This involves defining what constitutes normal behavior and flagging anything that deviates from it as an anomaly [38]. The challenge lies in accurately delineating the boundary of normal behavior, as it can be imprecise and subject to change. Additionally, the definition of anomalies can

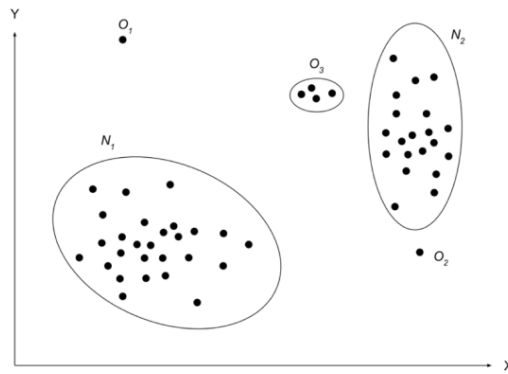


Figure 2.1: Illustration of simple anomalies in 2D

vary across different application domains, making it a complex task to develop universal detection techniques.

2.3 Anomaly Types

An essential aspect of anomaly detection is understanding the nature of the anomaly. Anomalies can be categorized in the following ways as shown in Figure. 2.2.

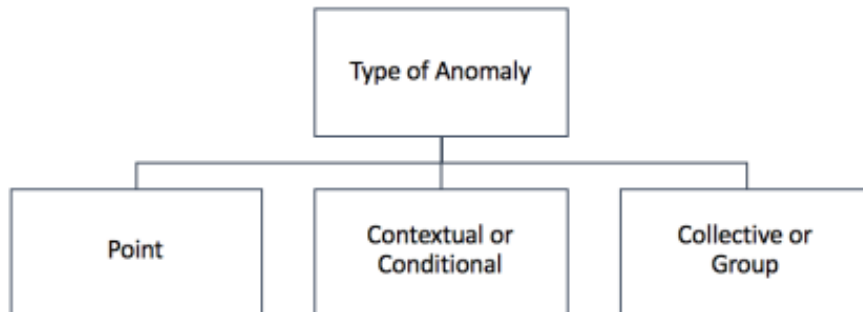


Figure 2.2: Anomaly Types [1]

- **Point anomaly:** Refers to a single data point that substantially deviates from the overall dataset [39]. In the context of the provided figure (Figure. 2.1), points o_1 , o_2 , and those falling within region o_3 are identified as point anomalies due to their positioning beyond the typical data distribution. Techniques for detecting these anomalies examine how an individual data instance relates to the rest of the dataset, whether it's part of the training data or the test data.
- **Contextual anomaly:** These anomalies occur when a data point is unusual within a specific context but appears normal otherwise. This concept, also known as a con-

ditional anomaly relies on the context defined by the dataset's structure, which must be clearly outlined in the problem formulation. Each data instance is characterized by two types of attributes:

- Contextual attributes: These attributes determine the context for a given instance. For example, in time-series data, time itself is a contextual attribute that situates an instance within the entire sequence [38].
- Behavioral attributes: These attributes describe the instance's non-contextual characteristics. For example, in a time series dataset that records daily CPU usage, the CPU measurements are the behavioral attributes [38].

Anomalous behavior is identified by examining the behavioral attributes within a specific context. A data instance might be deemed a contextual anomaly in one context but considered normal in another, despite having the same behavioral attributes. Recognizing this distinction is crucial for developing effective contextual anomaly detection techniques, as it hinges on accurately identifying both contextual and behavioral attributes. Choosing to use a contextual anomaly detection technique depends on how meaningful the contextual anomalies are for the specific application domain. Another crucial factor is whether contextual attributes are available. Sometimes, defining the context is straightforward, making it practical to apply contextual anomaly detection. However, in other situations, defining the context can be challenging, which complicates the use of these techniques.

- **Collective anomaly:** This type of anomalies occurs when a subset of data instances collectively deviate from the entire dataset. While individual instances in this subset may not be anomalies on their own, their combined occurrence forms an anomalous pattern [37]. These anomalies are significant particularly in data with spatial or sequential characteristics, manifesting as anomalous subgraphs or subsequences. For example, consider the following sequence of actions occurring on a computer: ... http-web, buffer-overflow, http-web, http-web, smtp-mail, ftp, http-web, ssh, smtp-mail, http-web, **ssh, buffer-overflow, ftp**, http-web, ftp, smtp-mail, http-web ... The highlighted sequence of events (ssh, buffer-overflow, ftp) represents a typical web-based attack by a remote machine, followed by data being copied from the host computer to a remote destination via FTP. This sequence is considered an anomaly, even though the individual events are not anomalies when they appear in other parts of the sequence.

2.4 Nature of Input Data

A key component of any anomaly detection technique is the input data, where anomalies need to be identified. This input typically consists of a collection of data objects or instances, which can also be referred to as records, points, vectors, patterns, events, cases, samples, observations, or entities [38]. Each data instance is described by a set of attributes, also known as variables, characteristics, features, fields, or dimensions. These instances can be of different types, such as binary, categorical, or continuous. A data instance might consist of a single attribute (univariate) or multiple attributes (multivariate) [40]. In multivariate instances, all attributes might be of the same type or a mixture of different types [41].

Input data can also be categorized based on the structure among data instances. Many anomaly detection algorithms handle data with no assumed structure, referred to as point data. Such algorithms are commonly used in network intrusion detection and medical records anomaly detection. However, data can also have spatial or sequential structures. In sequential data, instances have a defined order, with time-series data being a prime example. Time-series data has been extensively studied for anomaly detection in statistics.

2.5 Output of Anomaly Detection

The way anomalies are reported is a crucial aspect of any anomaly detection technique. Typically, the outputs fall into one of two categories:

- **Scores:** Scoring techniques assign an anomaly score to each instance in the test data, indicating how much it deviates from normal behavior. This results in a ranked list of anomalies, allowing analysts to either focus on the top anomalies or apply a cutoff threshold to select the most relevant ones. These techniques provide flexibility by letting analysts use domain-specific thresholds to highlight the most significant anomalies.
- **Labels :** Labeling techniques classify each test instance as either normal or anomalous. While this provides a straightforward indication of whether an instance is an anomaly, it lacks the granularity of scoring techniques. Analysts can't directly adjust the sensitivity of anomaly detection with labels, but they can influence it indirectly by tweaking parameters within the detection algorithm.

Both methods have their uses, with scoring techniques offering more detailed insights and labeling techniques providing clear, binary classifications.

2.6 Anomaly Detection Techniques

There are numerous methods for detecting anomalous events in time series data, ranging from basic arithmetic techniques to advanced neural networks such as Long Short-Term Memory (LSTM) autoencoders.

- **Fixed Threshold Method**

A fixed threshold method involves setting a predetermined value (e.g., CPU usage $> 80\%$ or $< 20\%$) beyond which an alert is triggered. While this approach is easy to configure and can sometimes be useful, it does not adapt to changing patterns over time. What might be an anomaly during peak hours may be normal during off-peak hours. Additionally, this method does not consider the context or trends in the data, meaning a sudden spike might be normal during certain periods but flagged as an anomaly if only the threshold is used. Consequently, fixed thresholds can lead to a high number of false positives (normal behavior flagged as anomalies) or false negatives (actual anomalies not detected).

- **Statistical Methods**

One popular technique for anomaly detection in time series data is the use of statistical methods. These methods utilize statistical properties of the data, such as the mean, standard deviation, and distributional assumptions, to identify points that significantly deviate from expected behavior. Common approaches include Z-score-based methods and moving average-based methods.

Consider the **Z-score** method for anomaly detection:

The Z-score method for anomaly detection measures how many standard deviations an observation is from the mean. An observation is considered an anomaly if its Z-score exceeds a certain threshold (e.g., $Z > 3$). It is calculated as $Z = \frac{X - \mu}{\sigma}$, where X is the data point, μ is the mean, and σ is the standard deviation. This method, known as the Median Absolute Deviation (MAD) based Z-score method, is useful in scenarios where data follows a normal distribution, such as detecting anomalies in manufacturing processes. However, the Z-score method assumes normality, which may not hold true for all datasets, making it less effective for skewed distributions [42] as explained in Figure. 2.5. It also relies on a fixed window size for calculating mean and standard deviation, potentially missing long-term trends or seasonal patterns. Additionally, it can be sensitive to outliers, which can skew the results and lead to inaccurate anomaly detection.

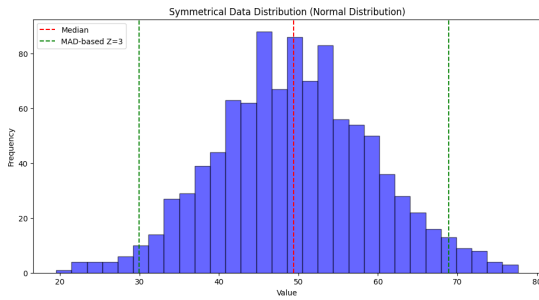


Figure 2.3: Normal Data Distribution

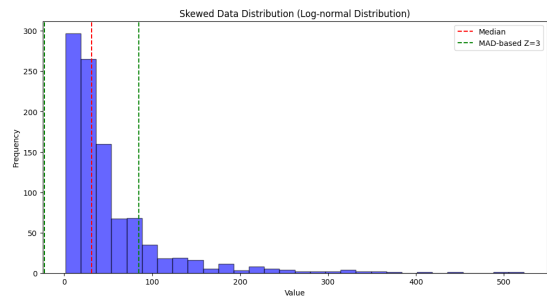


Figure 2.4: Skewed Data Distribution

Figure 2.5: The Influence of Data Distribution on the Z-score Method

The analysis of the graphs in Figure. 2.5 illustrates the effectiveness of the MAD-based Z-score method in detecting outliers in both normal and skewed distributions. In normal distributions, where data is evenly spread around the median, the MAD accurately reflects the data's dispersion. Consequently, outliers are properly identified as they significantly deviate from the median, as indicated by the threshold for $Z=3$. However, in skewed distributions, where the data is not evenly spread, the MAD underestimates the spread, particularly on the side with the long tail. This results in high values potentially not being detected as outliers, as the MAD fails to accurately represent the variability on the skewed side.

- **Machine learning-based Algorithms**

Machine Learning is a Subset of Artificial Intelligence (AI) that allows systems to independently learn and enhance their performance without explicit programming. Machine learning algorithms identify patterns in data and make predictions when new data is introduced to the system.

In Machine Learning, in addition to the input data, a dataset can include labels that indicate whether a data instance is normal or an anomaly. Based on how these labels are utilized, anomaly detection techniques can be categorized into three types:

- **Supervised Techniques:** These techniques rely on having a labeled training dataset that includes instances of both normal and anomaly classes. The typical approach involves building predictive models for each class. Any new data instance is then compared against these models to determine its classification. Supervised anomaly detection techniques benefit from a clear definition of normal and anomaly behaviors, enabling the construction of accurate models. However, a significant drawback is the high cost and effort required to obtain accurately labeled training data, as this process is often manual and involves

expert input. To address this, some techniques introduce artificial anomalies into a normal dataset to create a fully labeled training dataset [37], [38].

- **Semi-Supervised Techniques:** Methods operating in a semi-supervised manner work with training data containing labeled instances only for the normal class. This flexibility makes them more broadly applicable compared to supervised methods since they don't demand labels for the anomaly class [37], [38].
- **Unsupervised Techniques:** these techniques, operating without the need for labeled data, offer broad applicability, assuming that normal instances greatly outnumber anomalies in the test data. However, if this assumption doesn't hold true, these methods may produce a high false alarm rate [37], [38].

- **Deep Learning-based Algorithms**

Deep learning, a subset of machine learning, achieves high performance and flexibility by representing data through a nested hierarchy of concepts within the layers of a neural network. As illustrated in Figure. 2.6, deep learning outperforms traditional machine learning, particularly as the scale of data increases. In recent years, deep learning-based anomaly detection algorithms have gained popularity and have been applied to a diverse range of tasks. Specifically, for detecting anomalies in time series data, LSTM autoencoders have shown superior results compared to traditional autoencoders.

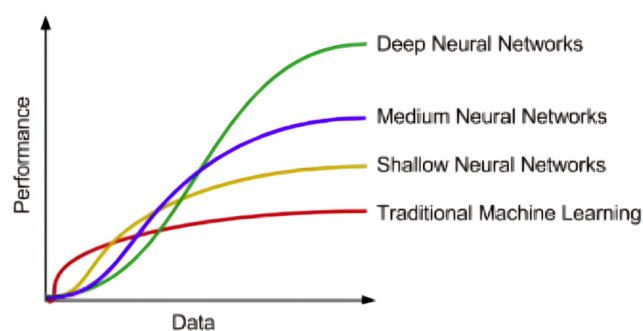


Figure 2.6: Performance Comparison of Deep learning-based algorithms Vs Traditional Algorithms [1]

Consider the **LSTM Autoencoder** algorithm for anomaly detection in time series data:

An autoencoder is an unsupervised neural network designed to learn the best encoding-decoding scheme from data. Its structure typically includes an input layer, an output layer, an encoder, a decoder, and a latent space [43]. The encoder compresses the input data into the latent space, while the decoder reconstructs the encoded data back to the output layer. The reconstructed output is then compared to the original input, and the error is backpropagated through the network to update the weights (weights are the neural network's way of learning from data. They capture the relationships between input features and the target output).

Given an input $x \in \mathbb{R}^m$, the encoder compresses it to an encoded representation $z = e(x) \in \mathbb{R}^n$. The decoder then reconstructs this representation to produce the output $\hat{x} = d(z) \in \mathbb{R}^m$ as shown in Figure. 2.7 which provides an illustration of a LSTM Autoencoder network [2]. The autoencoder is trained by minimizing the reconstruction error [44], [45] defined as:

$$L = \frac{1}{2} \sum_x \|x - \hat{x}\|^2.$$

The primary goal of an autoencoder is not only to copy the input to the output. By constraining the latent space to have a smaller dimension than the input (i.e., $n < m$), the autoencoder is compelled to learn the most significant features of the training data. Thus, a crucial aspect of an autoencoder is its ability to reduce data dimensions while preserving the essential information of the data structure.

The LSTM autoencoder uses Long Short-Term Memory (LSTM) networks for both the encoder and decoder. LSTMs which are type of Recurrent Neural Network (RNN), are adept at learning patterns in data over long sequences, making them suitable for tasks like time series forecasting and anomaly detection. An encoder-decoder model trained exclusively on normal sequences can be effective for anomaly detection in time-series data. Since the model learns to reconstruct only normal instances, it will have higher reconstruction errors for anomalous sequences, which it has not encountered during training. This approach is practical because anomalous data are often rare or impossible to cover comprehensively.

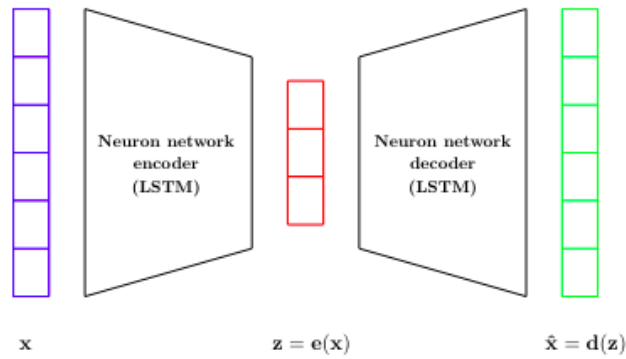


Figure 2.7: An illustration of a LSTM Autoencoder network [2]

LSTM-based autoencoders have shown superior results in anomaly detection compared to traditional autoencoders [2]. However, achieving significant improvements in detection accuracy with LSTM-based autoencoders often requires complex networks with large memory requirements and high computational complexity.

2.7 Anomaly Detection Process

Every step in the process of anomaly detection is crucial, starting with the analysis of the input data and the types of anomalies to detect. This involves understanding the nature of the data, the characteristics of the anomalies, and other constraints and assumptions that collectively form the problem formulation. The application domain in which the technique is applied is also important; while some techniques are developed in a more generic fashion and are feasible across multiple domains, others are specifically tailored to particular application domains. In the case of time series data, considering the contextual attribute of time dependence is essential for selecting the right techniques. Combining all these factors is critical to selecting the appropriate technique or techniques for effective anomaly detection, as illustrated in Figure 2.8.

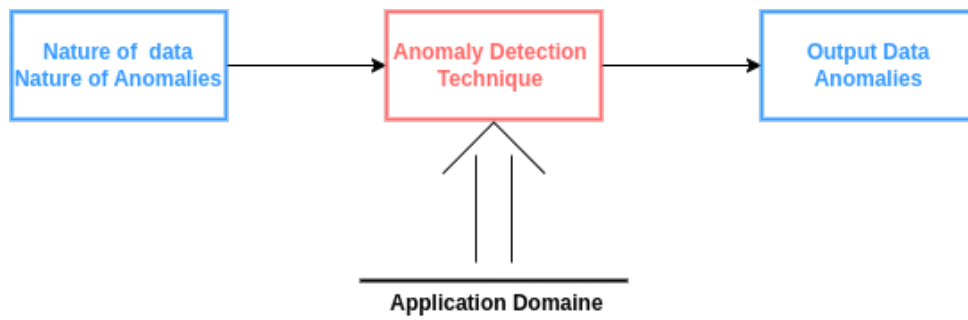


Figure 2.8: Anomaly Detection Process

2.8 Conclusion

In summary, detecting anomalies in time series data is essential for ensuring data accuracy and system reliability. By understanding the types of anomalies and the methods to identify them, we can better manage and respond to unexpected events in IT infrastructure monitoring.

Chapter 3

Conception and Implementation

3.1 Introduction

In this chapter, we present the conception and detailed implementation of our project, with a primary focus on developing a robust monitoring system. This includes deploying Prometheus for comprehensive metric collection, integrating exporters to capture specific metrics, and instrumenting web applications for enhanced monitoring capabilities. Furthermore, we describe the establishment of an alerting mechanism within Prometheus and the utilization of Grafana for insightful data visualization.

Moreover, this chapter explores the implementation of anomaly detection techniques tailored for analyzing time series data. We discuss the application of advanced methods, such as LSTM autoencoders, which are capable of capturing intricate patterns in time series data. These techniques are particularly suitable for detecting subtle anomalies that traditional statistical methods may overlook, thereby enhancing the reliability and effectiveness of our monitoring system.

By detailing the integration of these components and techniques, this chapter underscores the importance of a well-engineered monitoring system in ensuring the reliability and performance of complex systems. The practical implementation aspects, including deployment strategies using Docker Compose for scalability and management, are also highlighted.

3.2 Exploited Resources

As part of our work, we use specific Hardware resources and software tools selected for their compatibility with our needs to create a monitoring system within a Network architecture.

3.2.1 Hardware Resources

Processor : 1.6 GHZ Intel Core i5 (dual-core), Random Access Memory (RAM) : 16GO
2400 MHZ DDR4

3.2.2 Software Tools

For our project, we utilize Graphical Network Simulator-3 (GNS3) as a software tool to design a network architecture involving specific virtual machines created in VirtualBox. Additionally, we use the Graphical Network Simulator-3 Virtual Machine (GNS3 VM) to enable connections between these virtual machines. We also employ a Python virtual environment within our project to manage dependencies and installations. Furthermore, we deploy our project using Docker Compose to simplify management and ensure a consistent environment across different operating systems.

3.2.2.1 Graphical Network Simulator 3 - GNS3

GNS3 is an open-source graphical network simulator primarily used by network developers and IT professionals. Its main advantage is its ability to create visual representations of virtual topologies, providing an overview that facilitates real-time understanding of operations and allows for swift adjustments. Despite its capacity for handling complex tasks, the interface remains user-friendly.



Figure 3.1:
GNS3 Logo

3.2.2.2 Graphical Network Simulator-3 Virtual Machine - GNS3 VM

GNS3 VM is a virtual machine used in conjunction with the GNS3 network simulation software. It acts as a backend engine for GNS3, allowing users to create and simulate complex network topologies.

The GNS3 VM allows integration with VirtualBox by acting as an intermediary between VirtualBox and GNS3. This allows GNS3 to manage VirtualBox virtual machines as network devices within its simulated environments.

When the GNS3 VM is started, the screen shown in Figure. 3.2 appears, providing essential information for accessing and managing the VM and its network simulations.

3.2.2.3 VirtualBox

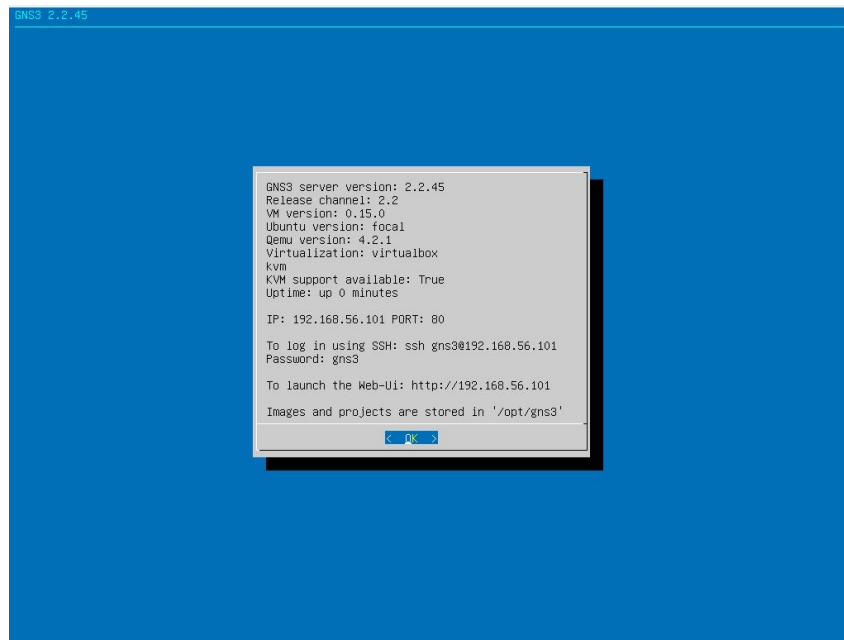


Figure 3.2: Initial Setup Screen of GNS3 VM

VirtualBox is an open-source virtualization tool that operates across various platforms developed by Oracle Corporation. It enables the simultaneous execution of multiple operating systems on a single device. Developers utilize VirtualBox to expedite their workflow by testing code across different operating systems directly from their laptops. This software is compatible with several operating systems, including Linux, Windows, and macOS.



Figure 3.3: Virtualbox Logo

3.2.2.4 Python Virtual Environment

Python virtual environment is an isolated space designated for Python projects, separate from the system-installed Python. It allows for setting up specific libraries and dependencies without impacting the system's Python installation. To create a Python virtual environment, we utilize a module called `virtualenv`. This tool generates a directory that includes all the essential executable required for using packages within a Python project. On Linux, we can create a new Virtual environment with the command:

```
\$ python3 -m venv <myenv>
```

Once the virtual environment is created, it must be activated. Activation of the appropriate virtual environment is necessary each time work is conducted on the project. This can be accomplished using the following command:

```
\$ source <myenv>/bin/activate
```

3.2.2.5 Docker and Docker Compose

Docker is an open platform for developing, shipping, and running applications. It allows to separate applications from the infrastructure, enabling faster software delivery and consistent management of infrastructure and applications. Docker's methodologies help reduce the delay between writing code and running it in production.



Figure 3.4: Docker Logo

Docker Compose is a tool used to define and run applications that involve multiple containers. It plays a crucial role in enhancing efficiency in development and deployment by managing containers as a unified application stack. This approach ensures consistent deployment across diverse environments. Docker Compose simplifies the management of the entire application stack by controlling services, networks, and storage volumes through a unified YAML configuration file **docker-compose.yml**. All services defined in this file can be created and started with a single command.

3.2.2.6 CAdvisor

CAvvisor (Container Advisor) provides container users with insights into the resource usage and performance characteristics of their running containers. It is a running daemon that collects, aggregates, processes, and exports information about these containers.



Figure 3.5: CAvvisor Logo

CAvvisor itself does not have built-in storage for the metrics it collects. Instead, it is primarily used for real-time monitoring. We integrate it with Prometheus, which can scrape and store the data provided by cAdvisor, allowing for historical data analysis, querying, and alerting.

3.3 Network Architecture

To create the network architecture shown in Figure 3.6, we first install GNS3 and VirtualBox. During the creation of the GNS3 VM, we allocate 2GB of RAM and 1 processor core. The network architecture consists of three virtual machines acting as servers: Kali Linux, Ubuntu, and Windows 10, with the specifications shown in table 3.1. Additionally, we include a Cisco IOU Layer 2 switch and a Cisco 7200 router in the setup. To ensure that the virtual machines can access the internet, we integrate a Network Address Translation (NAT) node into the architecture.

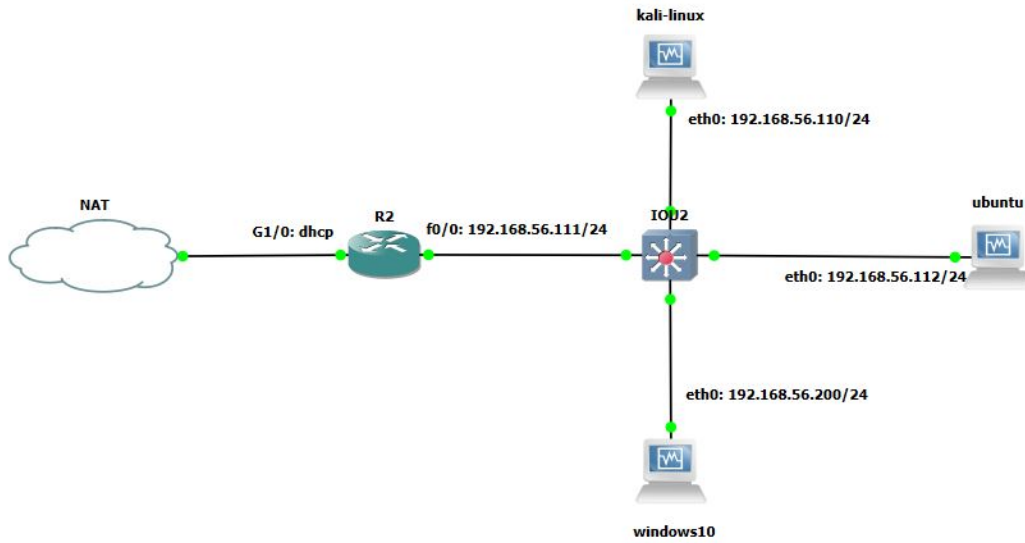


Figure 3.6: Network Architecture

Server	RAM	Processor	Disk Space	IP Address
Kali Linux	3 GB	3 processor	80 GB	192.168.56.110/24
Ubuntu	2 GB	1 processor	20 GB	192.168.56.112/24
Windows 10	2 GB	1 processor	20 GB	192.168.56.200/24

Table 3.1: Server Specifications

The network is designed to provide an isolated internal network (192.168.56.0/24). The router R2 acts as the gateway for all internal hosts to access external resources. The IOU2 switch connects all internal hosts within the subnet 192.168.56.0/24. The configuration of the router R2 and the switch is shown in Figure. 3.7 and Figure. 3.8 respectively.

3.4 Implementation of The Monitoring System

In our project, we first deploy the monitoring system locally on the Kali Virtual Machine (VM). Subsequently, we use Docker-Compose to deploy the system, ensuring compatibility across various environments regardless of operating system types or versions. This section reviews the implementation of Prometheus, exporters, alerting mechanism and Grafana as well as the instrumentation of a web application.

The design and implementation of the monitoring system are shown in Figure. 3.9, where all the tools are integrated to design the system. The configuration of each tool is detailed in the following sections.


```

R1#en
R1#config t
Enter configuration commands, one per line. End with CNTL/Z.
R1(config)#int f0/0
R1(config-if)#ip add 192.168.56.111 255.255.255.0
R1(config-if)#ip nat inside
R1(config-if)#no shut
R1(config-if)#exit
R1(config)#access-list 9 permit 192.168.56.0 0.0.0.255
R1(config)#ip nat source list 9 int g1/0 overload
R1(config)#int g1/0
R1(config-if)#ip add dhcp
R1(config-if)#ip nat ou
R1(config-if)#ip nat ou
*Jun 13 22:53:52.507: %DHCP-6-ADDRESS_ASSIGN: Interface GigabitEthernet1/0 assigned DHCP address 192.168.13.131, mask 255.255.255.0, hostname R1

R1(config-if)#ip nat outside
R1(config-if)#no shut
R1(config-if)#ping 8.8.8.8
^
% Invalid input detected at '^' marker.

R1(config-if)#end
R1#
*Jun 13 22:54:48.703: %SYS-5-CONFIG_I: Configured from console by console
R1#write memory
Warning: Attempting to overwrite an NVRAM configuration previously written
by a different version of the system image.
Overwrite the previous NVRAM configuration?[confirm]y
Building configuration...
[OK]
R1#
R1#ping 8.8.8.8
Type escape sequence to abort.
Sending 5, 100-byte ICMP Echos to 8.8.8.8, timeout is 2 seconds:
!!!!
Success rate is 100 percent (5/5), round-trip min/avg/max = 40/65/96 ms
R1#
    
```

Figure 3.7: Network Configuration of the Router

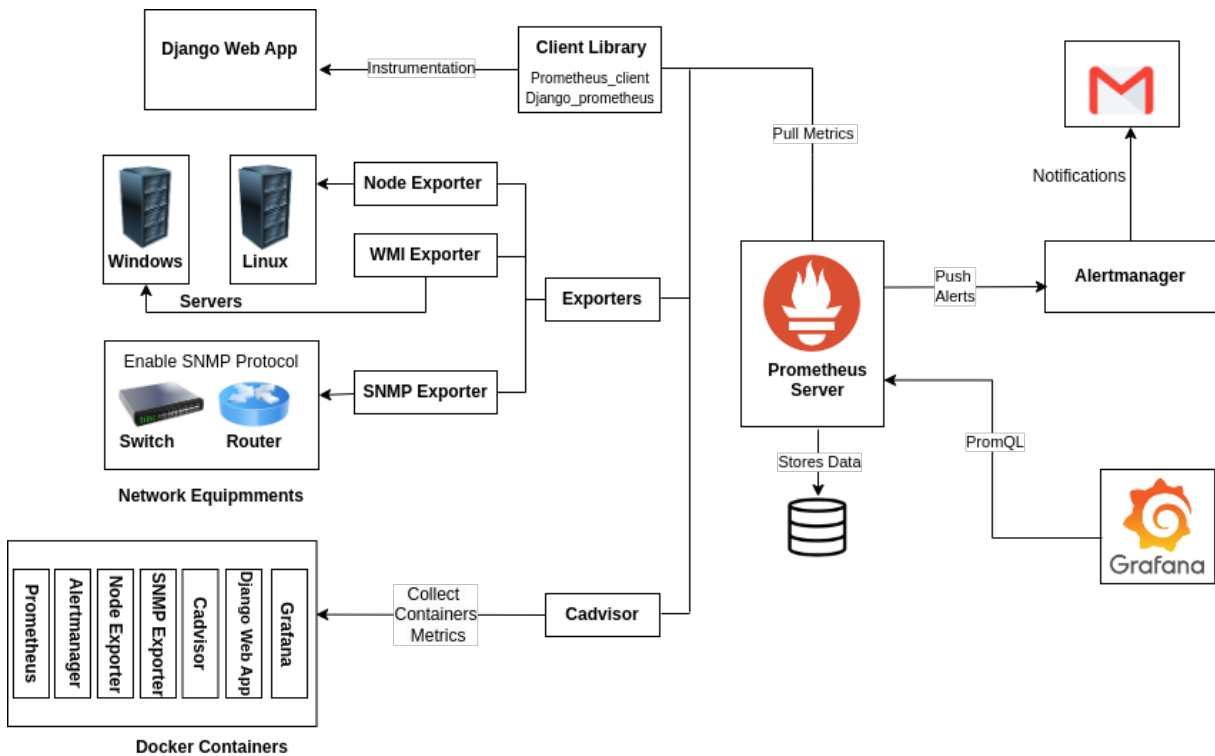


Figure 3.9: Conception and implementation of the monitoring system

```
IOU2#en
IOU2#config t
Enter configuration commands, one per line. End with CNTL/Z.
IOU2(config)#int vlan1
IOU2(config-if)#ip add 192.168.56.113 255.255.255.0
IOU2(config-if)#ip default-gateway 192.168.56.111
IOU2(config)#int e0/0
IOU2(config-if)#switchport mode access
IOU2(config-if)#exit
IOU2(config)#int e1/0
IOU2(config-if)#switchport mode access
IOU2(config-if)#exit
IOU2(config)#int e0/1
IOU2(config-if)#switchport mode access
IOU2(config-if)#exit
IOU2(config)#int e0/2
IOU2(config-if)#switchport mode access
IOU2(config-if)#exit
IOU2(config)#int e0/3
IOU2(config-if)#switchport mode access
IOU2(config-if)#exit
IOU2(config)#end
IOU2#
*Jun 13 19:08:55.663: %SYS-5-CONFIG_I: Configured from console by console
IOU2#write memory
Building configuration...
Compressed configuration from 1662 bytes to 1006 bytes[OK]
IOU2#
```

Figure 3.8: Network Configuration of the Switch

3.4.1 Prometheus

We begin by installing the monitoring tool Prometheus on the Kali VM. Prometheus implemented as a executable file and is accompanied by several files, most notably `prometheus.yml`, which serves as the configuration file of prometheus.

In this configuration file, we define the scrape interval and evaluation interval as 15 seconds. Additionally, we specify the paths to rules files and configure the alerting mechanism as shown in Figure. 3.10. Furthermore, we define our targets for Prometheus, specifying each target's address and the appropriate port to access their metrics endpoint as shown in Figure. 3.11 in the `scrape_configs` section.

The targets we are monitoring include:

- Prometheus: It monitors its own metrics by default.

- Grafana: Monitor the statue of Grafana server.
- The servers: Kali, Ubuntu, Windows10.
- Django web application: Monitored for performance metrics.
- CAdvisor: Monitors container statuses.
- SNMP exporter: Provides metrics about its statue.
- Target network: Includes the network equipment, which are switch and router.

```
# my global config
global:
  scrape_interval: 15s # Set the scrape interval to every 15 seconds. Default is every 1 minute.
  evaluation_interval: 15s # Evaluate rules every 15 seconds. The default is every 1 minute.
  # scrape_timeout is set to the global default (10s).

# Alertmanager configuration
alerting:
  alertmanagers:
    - static_configs:
      - targets:
        - alertmanager:9093

# Load rules once and periodically evaluate them according to the global 'evaluation_interval'.
rule_files:
- "alerting_rules/linuxrules.yml"
- "alerting_rules/windowsrules.yml"
- "alerting_rules/apprules.yml"
- "recording_rules/rules.yml"

# A scrape configuration containing exactly one endpoint to scrape:
# Here it's Prometheus itself.
scrape_configs:
  # The job name is added as a label `job=<job_name>` to any timeseries scraped from this config.

  - job_name: "Prometheus"
    static_configs:
      - targets: ["prometheus:9090"]

  - job_name: "Grafana"
    static_configs:
      - targets: ["grafana:3000"]

  - job_name: "VM-Kali"
    file_sd_configs:
      - files:
        - file_sd/file_sd1.yml

  - job_name: "VM-Windows10"
    static_configs:
      - targets: ["192.168.56.200:9182"]

  - job_name: "VM-Ubuntu"
    file_sd_configs:
      - files:
        - file_sd/file_sd2.yml
```

Figure 3.10: Prometheus Configuration File 1

```
- job_name: "django-app"
  static_configs:
    - targets: ["web:8000"]

- job_name: 'cadvisor'
  static_configs:
    - targets: ['cadvisor:8080']

- job_name: "snmp-exporter"
  static_configs:
    - targets: ["snmp-exporter:9116"]

- job_name: "network"
  static_configs:
    - targets:
      - '192.168.56.111'
      - '192.168.56.113'
  metrics_path: /snmp
  params:
    module: [standard_mibs]
  relabel_configs:
    - source_labels: [__address__]
      target_label: __param_target
    - source_labels: [__param_target]
      target_label: instance
    - source_labels: [__address__]
      regex: "192.168.56.111"
      target_label: __param_auth
      replacement: "router"
    - source_labels: [__address__]
      regex: "192.168.56.113"
      target_label: __param_auth
      replacement: "switch"
    - target_label: __address__
      replacement: snmp-exporter:9116
```

Figure 3.11: Prometheus Configuration File 2

Prometheus provides a tool called **promtool** which is used to validate and check the correctness of Prometheus configuration files as shown in Figure. 3.12 Promtool can also check rules for validating alerting and recording rules using the command:

```
\$ ./promtool check rules.yml
```

```
(kali@kali)-[~/Documents/monitoring/prometheus]
└─$ ./promtool check config prometheus.yml
Checking prometheus.yml
  SUCCESS: 4 rule files found
  SUCCESS: prometheus.yml is valid prometheus config file syntax

Checking alerting_rules/linuxrules.yml
  SUCCESS: 5 rules found

Checking alerting_rules/windowsrules.yml
  SUCCESS: 5 rules found

Checking alerting_rules/apprules.yml
  SUCCESS: 1 rules found

Checking recording_rules/rules.yml
  SUCCESS: 5 rules found
```

Figure 3.12: Checking Prometheus.yml file using Promtool

3.4.1.1 Recording Rules

In our recording rules shown in Figure. 3.13, we create three groups: one for Linux servers, one for Windows servers, and one for the web application. We define rules within each group based on the jobs. For each group, we specify the expressions to aggregate, written in PromQL, and assign new names to these expressions. It is important to follow the naming conventions for recording rules, which follow the format: **level:metric:operation**. We choose these metrics because they are frequently used and regularly monitored.

```
groups:
- name: Linux
  rules:
  - record: job:node_cpu_usage:percentage
    expr: 100 * (1 - avg(rate(node_cpu_seconds_total{mode="idle"}[5m])) by (instance))
  - record: job:node_memory_usage:percentage
    expr: 100 * (1 - avg((node_memory_MemFree_bytes / node_memory_MemTotal_bytes)) by (instance))

- name: Windows
  rules:
  - record: job:windows_cpu_usage:percentage
    expr: 100 * avg without (cpu) (sum(irate(windows_cpu_time_total{mode="idle"}[5m])) by (instance))
  - record: job:windows_memory_usage:percentage
    expr: 100 * (1 - avg((windows_os_physical_memory_free_bytes / windows_cs_physical_memory_bytes)) by (instance))

- name: DjangoApp
  rules:
  - record: job:django_http_requests_rate:5m
    expr: rate(django_http_requests_total_by_method_total{instance=~"$instance",job="django-app"}[5m])
  - record: job:django_memory_usage:percentage
    expr: (sum(process_resident_memory_bytes{instance=~"$instance",job="django-app"}) / sum(node_memory_MemTotal_bytes{job="VM-Kali"})) * 100
```

Figure 3.13: Recording Rules file rules.yml

3.4.1.2 File-based Service Discovery

We apply File-based Service Discovery to two targets, Kali Linux and Ubuntu as shown in Figure. 3.14, to observe the differences between using File-based Service Discovery and static configuration. However, our goal is to extend this approach to all targets. By doing so, any changes in the configuration of these targets, such as adding labels or changing IP addresses, will be automatically applied without needing to restart the Prometheus server. This ensures seamless updates and improved efficiency in managing our monitoring setup.

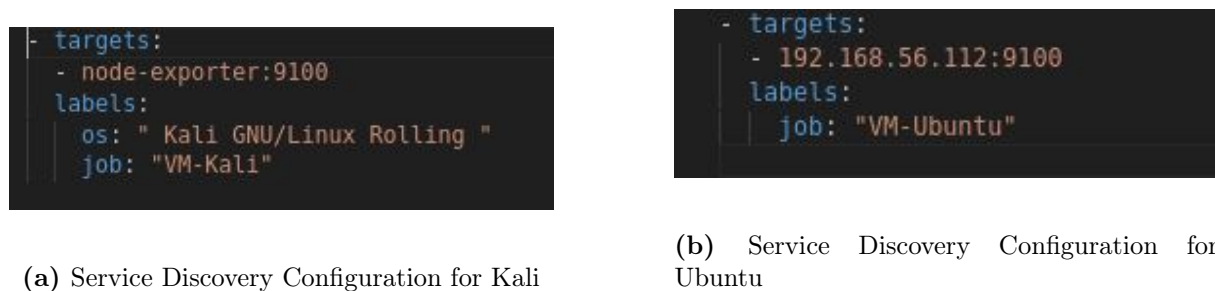


Figure 3.14: File-based Service Discovery

3.4.2 Exporters

To effectively collect system metrics, it is necessary for the target devices to expose their metrics through an HTTP endpoint. This is achieved by installing exporters on the target devices.

3.4.2.1 Node Exporter

It's crucial to configure the exporter to run as a service, ensuring it automatically restarts upon system reboots for continuous operation without manual intervention. Here's how we achieve this for the Node Exporter:

Create a Systemd Service File with the command:

```
\$ sudo nano /etc/systemd/system/node_exporter.service
```

Add the content shown in Figure. 3.15 to the file.

Reload the systemd daemon to recognize the new service, Start the Node Exporter service and enable it to start on boot:

```
\$ sudo systemctl daemon-reload
\$ sudo systemctl start node_exporter
\$ sudo systemctl enable node_exporter
```

```

ubuntu@ubuntu-VirtualBox: /etc/systemd/system
GNU nano 4.8 node_exporter.service
[Unit]
Description=Node_exporter server
After=network.target

[Service]
User=ubuntu
Group=ubuntu
Type=simple
WorkingDirectory=/usr/local/bin/node_exporter
ExecStart=/usr/local/bin/node_exporter/node_exporter

[Install]
WantedBy=multi-user.target

```

Figure 3.15: Content of Systemd Service File for Node Exporter

check the status of the Node Exporter service :

```
\$ sudo systemctl status node_exporter
```

We can see that the Node Exporter service is active and running as illustrated in Figure. 3.16

```

ubuntu@ubuntu-VirtualBox:/etc/systemd/system$ sudo systemctl status node_exporter.service
● node_exporter.service - Node_exporter server
   Loaded: loaded (/etc/systemd/system/node_exporter.service; enabled; vendor preset: enabled)
   Active: active (running) since Thu 2024-05-23 20:46:19 EDT; 21min ago
     Main PID: 6622 (node_exporter)
        Tasks: 6 (limit: 2276)
       Memory: 9.1M
      CGroup: /system.slice/node_exporter.service
             └─6622 /usr/local/bin/node_exporter/node_exporter

May 23 21:05:45 ubuntu-VirtualBox node_exporter[6622]: ts=2024-05-24T01:05:45.
May 23 21:06:00 ubuntu-VirtualBox node_exporter[6622]: ts=2024-05-24T01:06:00.
May 23 21:06:15 ubuntu-VirtualBox node_exporter[6622]: ts=2024-05-24T01:06:15.
May 23 21:06:30 ubuntu-VirtualBox node_exporter[6622]: ts=2024-05-24T01:06:30.
May 23 21:06:45 ubuntu-VirtualBox node_exporter[6622]: ts=2024-05-24T01:06:45.
May 23 21:07:00 ubuntu-VirtualBox node_exporter[6622]: ts=2024-05-24T01:07:00.
May 23 21:07:15 ubuntu-VirtualBox node_exporter[6622]: ts=2024-05-24T01:07:15.
May 23 21:07:30 ubuntu-VirtualBox node_exporter[6622]: ts=2024-05-24T01:07:30.
May 23 21:07:45 ubuntu-VirtualBox node_exporter[6622]: ts=2024-05-24T01:07:45.
May 23 21:08:00 ubuntu-VirtualBox node_exporter[6622]: ts=2024-05-24T01:08:00.
lines 1-19/19 (END)

```

Figure 3.16: Node Exporter Service Running

3.4.2.2 WMI Exporter

We use Non-Sucking Service Manager (NSSM) to configure the WMI Exporter to operate as a service. First, we download NSSM and extract its contents from the ZIP file. Subsequently, we execute the following commands in Command Prompt or PowerShell with administrative privileges:

```
> C:\Users\windows10\Downloads\nssm-2.24\nssm-2.24\win64\nssm install
↳ windows_exporter C:\Users\windows10\Downloads\windows_exporter.exe
```

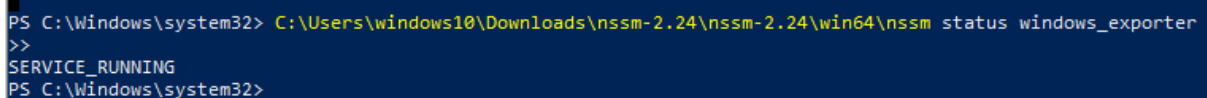
```
> C:\Users\windows10\Downloads\nssm-2.24\nssm-2.24\win64\nssm set
↳ windows_exporter Start SERVICE_AUTO_START
```

```
> C:\Users\windows10\Downloads\nssm-2.24\nssm-2.24\win64\nssm start
↳ windows_exporter
```

To verify if the WMI Exporter service is running, execute the command:

```
> C:\Users\windows10\Downloads\nssm-2.24\nssm-2.24\win64\nssm status
↳ windows_exporter
```

The results can be observed as depicted in Figure. 3.17.



```
PS C:\Windows\system32> C:\Users\windows10\Downloads\nssm-2.24\nssm-2.24\win64\nssm status windows_exporter
>>
SERVICE_RUNNING
PS C:\Windows\system32>
```

Figure 3.17: WMI Node Exporter Service Running

3.4.2.3 SNMP Exporter

Here is our approach to monitor network devices using SNMP exporter:

- We configure SNMP protocol v3 on both the router and the switch. SNMP Version 3 provides secure access to devices by authenticating and encrypting data packets over the network.

In Figure. 3.18, we show the configuration of SNMP v3 on the router, including the setup of the SNMP group, SNMP user, authentication protocol, and privacy protocol. Similarly, Figure. 3.19 displays the SNMP v3 configuration for the switch. Additionally, we specify the SNMP host server by providing its IP address, as shown in the right side of Figure. 3.19.


```
R1#
R1#en
R1#config t
Enter configuration commands, one per line. End with CNTL/Z.
R1(config)#snmp-server group snmpgrp v3 ?
  auth    group using the authNoPriv Security Level
  noauth  group using the noAuthNoPriv Security Level
  priv    group using SNMPv3 authPriv security level

R1(config)#snmp-server group snmpgrp v3 priv
R1(config)#snmp-server user snmpuser snmpgrp v3 ?
  access  specify an access-list associated with this group
  auth    authentication parameters for the user
  encrypted specifying passwords as MD5 or SHA digests
  <cr>

R1(config)#snmp-server user snmpuser snmpgrp v3 auth sha monitoring2024 ?
  access  specify an access-list associated with this group
  priv    encryption parameters for the user
  <cr>

R1(config)#$ user snmpuser snmpgrp v3 auth sha monitoring2024 priv aes 128
snmp-server user snmpuser snmpgrp v3 auth sha monitoring2024 priv aes 128
% Incomplete command.

R1(config)#$ user snmpuser snmpgrp v3 auth sha monitoring2024 priv aes 128 ?
  WORD   privacy password for user

R1(config)#$rp v3 auth sha monitoring2024 priv aes 128 monitoring2024
R1(config)#end
R1#
*Jun 13 23:08:39.947: %SYS-5-CONFIG_I: Configured from console by console
R1#write memory
Building configuration...
[OK]
R1#show snmp user

User name: snmpuser
Engine ID: 800000090300CA01033C0000
storage-type: nonvolatile      active
Authentication Protocol: SHA
Privacy Protocol: AES128
Group-name: snmpgrp
```

Figure 3.18: SNMP v3 Configuration on The Router

- After enabling SNMP protocol on both the router and the switch, we install the SNMP Exporter on the Kali machine to collect data from these two targets.
- Then, we create a new snmp.yml file, in which we define the SNMP version and the target devices for monitoring.

```
auths:
  router:
    version: 3
```

```
IOU2#en
IOU2#config t
Enter configuration commands, one per line. End with CNTL/Z.
IOU2(config)#snmp-server group snmpgrp v3 auth
IOU2(config)#$ user snmpuser snmpgrp v3 encrypted auth sha monitoring2024
%Error in Authentication password
IOU2(config)#snmp-server user snmpuser snmpgrp v3 encrypted auth sha monitoring$
%Error in Authentication password
IOU2(config)#snmp-server user snmpuser snmpgrp ?
  remote Specify a remote SNMP entity to which the user belongs
  v1      user using the v1 security model
  v2c     user using the v2c security model
  v3      user using the v3 security model

IOU2(config)#snmp-server user snmpuser snmpgrp v3 ?
  access specify an access-list associated with this group
  auth   authentication parameters for the user
  encrypted specifying passwords as MD5 or SHA digests
  <cr>

IOU2(config)#snmp-server user snmpuser snmpgrp v3 auth ?
  md5 Use HMAC MD5 algorithm for authentication
  sha Use HMAC SHA algorithm for authentication

IOU2(config)#snmp-server user snmpuser snmpgrp v3 auth sha ?
  WORD authentication password for user

IOU2(config)#snmp-server user snmpuser snmpgrp v3 auth sha monitoring2024
```

```
IOU2(config)#snmp-server host 192.168.56.110 v3
IOU2(config)#end
IOU2#wr
*Jun 13 19:38:24.159: %SYS-5-CONFIG_I: Configured from console by console
IOU2#write memory
Building configuration...
Compressed configuration from 1698 bytes to 1024 bytes[OK]
IOU2#show snmp user

User name: snmpuser
Engine ID: 800000090300AABBCC000100
storage-type: nonvolatile active
Authentication Protocol: SHA
Group-name: snmpgrp
```

Figure 3.19: SNMP v3 Configuration on The Switch

```
security_level: authPriv
auth_protocol: SHA
username: snmpuser
password: monitoring2024
priv_protocol: AES
priv_password: monitoring2024

switch:
  version: 3
  security_level: authNoPriv
  auth_protocol: SHA
  username: snmpuser
  password: monitoring2024
```

In this file, we also specify the data to collect. SNMP data is structured in OID trees, which are described by MIBs. OID subtrees maintain a consistent order across different locations in the tree. For example, the order under 1.3.6.1.2.1.2.2.1.1 (ifIndex) remains the same across different parts of the tree.

```
- name: ifIndex
  oid: 1.3.6.1.2.1.2.2.1.1
  type: gauge
  indexes:
    - labelname: ifIndex
      type: Integer
```

The full snmp.yml file is in [Appendix A.1](#).

- The final step is to add snmp exporter and network devices in prometheus configuration file as shown in [Figure 3.11](#).

3.4.3 Web Application Instrumentation

In this section, we expand our monitoring and data collection to include applications by instrumenting a Django web application. This involves integrating additional lines of code into the application source code to generate metrics reflecting its internal state.

We employ two distinct libraries to instrument the Django web application: the Django Prometheus library and the Prometheus client library.

The Django Prometheus library is tailored specifically for Django applications. It simplifies integration with Django's internals by providing pre-configured settings. Additionally, it automates the generation and exposition of metrics across different aspects of the Django application.

In contrast, the Prometheus client library offers extensive control and flexibility but demands more manual configuration. It supports multiple programming languages like Python, Go, Java, and JavaScript, allowing users to define custom metrics and create an HTTP endpoint for exposing these metrics.

To achieve this instrumentation, we upload the web application's codebase to the Kali server. The structure of the application is illustrated in Figure 3.20:

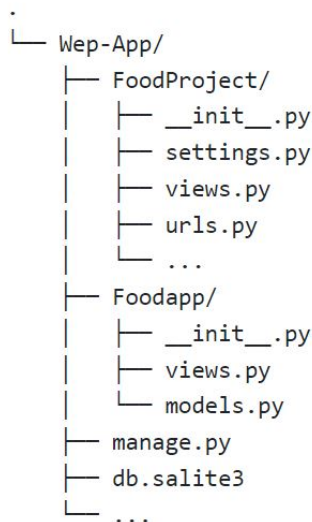


Figure 3.20: Tree Structure of the Uploaded Django Web Application

All changes are made within a Python virtual environment named **django-app**. In this environment, we install the required packages and run the web application in isolation. The web application is configured to run on localhost using port 8000. To access the application, we use the following command:

```
(django-app) kali@kali:\$ python3 manage.py runserver
```

When accessed, the home page of the web application appears as shown in Figure 3.21:

3.4.3.1 Web Application Instrumentation using Django Prometheus Library

To instrument the Django web application using Django Prometheus library, we follow these steps:

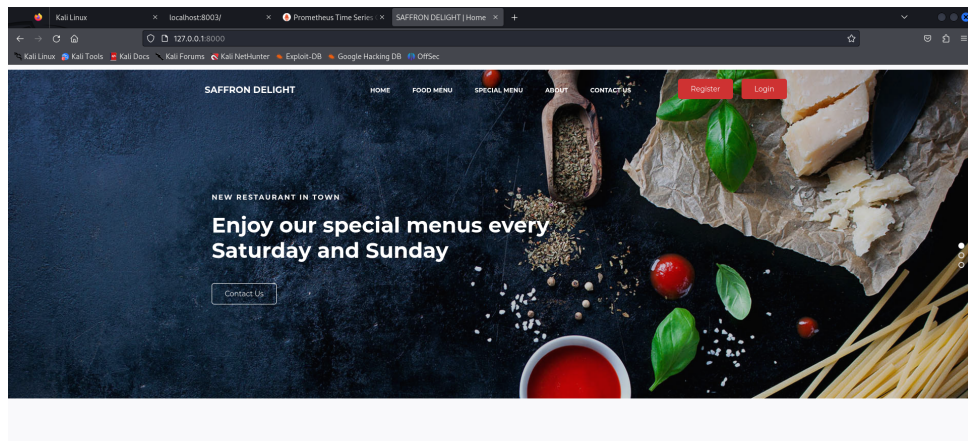


Figure 3.21: The Home Page of the Web Application

- Install `django-prometheus` with the command:

```
(django-app) kali@kali:\$ pip install django_prometheus
```

- Add the following lines to `settings.py` file:

```
INSTALLED_APPS = [  
    ...  
    'django_prometheus',  
    ...  
]  
MIDDLEWARE = [  
    'django_prometheus.middleware.PrometheusBeforeMiddleware',  
    ...  
    'django_prometheus.middleware.PrometheusAfterMiddleware',  
]
```

By adding `django_prometheus` to `INSTALLED_APPS`, Django recognizes the Django Prometheus library as part of the application setup. This allows Django to load and utilize functionalities provided by this library.

Prometheus middleware is configured both before and after other middleware classes to capture request duration and response duration, among other metrics.

- Include Django Prometheus in `urls.py`, this step enables the collection of metrics accessible via an HTTP endpoint at **`http://127.0.0.1:8000/metrics`** where Prometheus can scrape the metrics. This endpoint exposes metrics in a format that Prometheus can understand.

```
urlpatterns = [  
    ...  
    path('', include('django_prometheus.urls')),  
]
```

- we also enable database monitoring using Django Prometheus by modifying the `ENGINE` property of the database in `settings.py`, replacing `django.db.backends` with `django_prometheus.db.backends`, as shown in the following code:

```
DATABASES = {  
    'default': {  
        'ENGINE': 'django_prometheus.db.backends.sqlite3',  
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),  
    },  
}
```

- The final step involves configuring Prometheus to Scrape the metrics, in Prometheus configuration file `prometheus.yml`:

```
- job_name: "django-app"  
  static_configs:  
    - targets: ["127.0.0.1:8000"]
```

3.4.3.2 Web Application Instrumentation using Prometheus Client Library

As mentioned earlier, this method allows us to configure custom metrics as needed. In our case, we create two metrics: `django_sign_in_failures` and `django_registered_users`.

The first metric tracks the total number of failed sign-in attempts. It functions as a counter, incrementing with each failed attempt. This metric is useful for monitoring the health and security of the Web application. A high rate of sign-in failures might indicate issues such as users experiencing difficulties logging in due to a bug, or a potential

security threat, such as a brute-force attack. The second metric is created to monitor the total number of registered users in the web application. It enables administrators and developers to track the increase in users over time, providing insights into the application's popularity and user engagement.

Here is our procedure to instrument the Django web application using the Prometheus Client Library:

- Install the Prometheus Client Library by running the following command:

```
(django-app) kali@kali:\$ pip install prometheus_client
```

- Add `prometheus_client` to `INSTALLED_APPS` in `settings.py`:

```
INSTALLED_APPS = [  
    ...  
    'prometheus_client',  
    ...  
]
```

- Create a new file `metrics.py` in `Foodapp` where we configure the metrics as follows:

```
from prometheus_client import Counter  
  
sign_in_failures = Counter('django_sign_in_failures', 'Number of  
→ failed sign-in attempts')  
registered_users_counter = Counter('django_registered_users',  
→ 'Number of registered users')
```

- Update `views.py` which located in `Foodapp` directory, with the necessary changes to increment the counters for sign-in failures and new user registrations:

```
#apply to sign_in_failures and registered_users_counter metrics from  
→ metrics file  
from .metrics import sign_in_failures, registered_users_counter
```

```
def doLogin(request):
    if request.method=="POST":
        ...
        if utype == "Admin":
            ...
            else:
                sign_in_failures.inc()# add this line to increment
                ↪ the sign-in failure counter
                return
                ↪ render(request,"login.html",{ 'failure': 'Incorrect
                ↪ login details'})

        if utype == "User":
            ...
            else:
                sign_in_failures.inc() # add this line to increment
                ↪ the sign-in failure counter
                return
                ↪ render(request,"login.html",{ 'failure': 'Incorrect
                ↪ login details'})
            ...

def addcust(request):
    ...
    if form.is_valid():
        try:
            form.save()
            registered_users_counter.inc()# add this line to
            ↪ increment the number of registered users
            ↪ counter
            return redirect("/login")
        except:
            ...
```

In this code, we import the necessary metrics and utilize the `.inc()` function to

increment the counts of sign-in failures and newly registered users, as specified in the comments.

- Expose Metrics Endpoint by creating a `views.py` file in the `FoodProject` directory to expose the created metrics to Prometheus. Below is the snippet code to accomplish this :

```
from prometheus_client import exposition, REGISTRY
from django.http import HttpResponse

def metrics_view(request):
    response =
        ↪ HttpResponse(content_type=exposition.CONTENT_TYPE_LATEST)
    response.content = exposition.generate_latest(REGISTRY)
    return response
```

When the `metrics_view` function is called with an HTTP request, a new `HttpResponse` object is created with the appropriate content type for Prometheus metrics. The response content is then set to the latest metrics data generated by Prometheus Client Library, and the response is returned, allowing Prometheus to scrape the metrics.

- Add the `metrics_view` to the `/metrics` endpoint in the `urls.py` file:

```
urlpatterns = [

    path('metrics/', metrics_view, name='metrics'),

]
```

When a request is made to the URL `http://127.0.0.1:8000/metrics`, Django will call the `metrics_view` function to handle the request. Furthermore, there's no requirement to configure Prometheus for metric scraping because it's already integrated during the implementation of the web application's instrumentation using Django Prometheus.

3.4.3.3 Deployment of The Django Application with Docker Compose

After successfully setting up the Django application locally, it's time to use Docker. Docker provides an efficient way to package the Django application with its dependencies and configurations into a self-contained unit called a container.

Here are the steps we follow to deploy the Django web application with Docker Compose:

- Create a **Dockerfile** in the root of the application directory.

We create a Dockerfile that defines the instructions for building a Docker image. In this Dockerfile, we specify the base image, copy the application code, install dependencies, configure the container environment, and finally, specify the command to launch the application server. The following is the Dockerfile for our application.

```
FROM python:3.9-slim
# Set environment variables to ensure Python output is not buffered
ENV PYTHONDONTWRITEBYTECODE 1
ENV PYTHONUNBUFFERED 1

WORKDIR /app
COPY requirements.txt /app/
RUN pip install --no-cache-dir -r requirements.txt
COPY . /app/
RUN python manage.py collectstatic --noinput
CMD ["python", "manage.py", "runserver", "0.0.0.0:8000"]
```

- Add the following two libraries to the **requirements.txt** file:

```
prometheus_client==0.20.0
django_prometheus==2.3.1
```

- Create a **docker-compose.yml** file in the root of the application directory. In this file, we define the services needed for the application. Typically, there is a service for the application and another for the database. However, in our case, there is only one service **web**, which represents the app service. This is because the application uses SQLite, which is a file-based database and not a standalone database server.

Additionally, we specify the network which is the `monitoring` network and set it as external for monitoring purposes. The port to expose is configured as 8000:8000.

- The final step is to run the application using the following command to view all the logs and errors:

```
\$ sudo docker-compose up --build
```

Alternatively, run it as a daemon using:

```
\$ sudo docker-compose up -d
```

3.4.4 Establishing Alerting Mechanism in Prometheus

To establish alerting in Prometheus, we first create alerting rules in a new folder named ‘`alerting_rules`’. In this folder, we create separate files for each job, including the operating systems and the web application, to configure the alerting rules. We established four alerts for each operating system:

- An alert when the system is down.
- An alert when memory usage exceeds 60%, with a warning severity.
- An alert when memory usage exceeds 75% with critical severity.
- An alert when CPU usage exceeds 80%.

Additionally, we create an alert for the Django web application to trigger when the number of sign-in failures is high. The alerting rules are configured using YAML. The diagram below in Figure. 3.22 summarizes the configuration files for the alerting rules.

Each alert is set to trigger if the condition persists for more than 2 minutes. The labels and annotations are used to provide additional informations for each alert. After creating the alerting rules, we load them into the Prometheus configuration file using the `rule_files` directive:

```
rule_files:  
- "alerting_rules/linuxrules.yml"  
- "alerting_rules/windowsrules.yml"  
- "alerting_rules/apprules.yml"
```

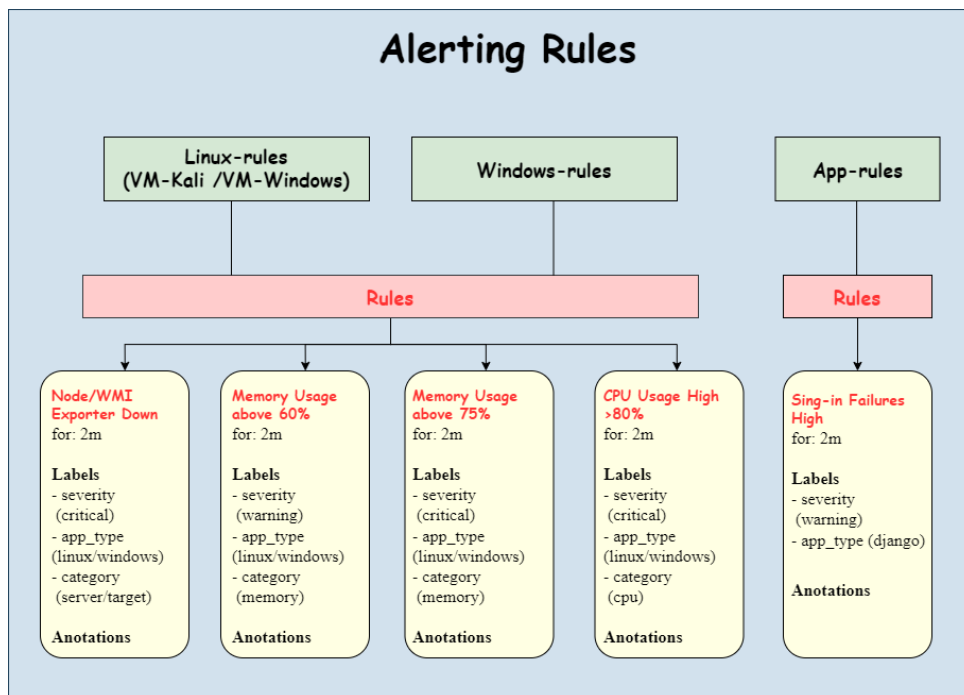


Figure 3.22: Diagram of Alerting Rules Implementation

The next step involves focusing on sending firing alerts within notifications. For this purpose, we install Alertmanager. By default, Alertmanager uses port 9093. Therefore, we configure it in the Prometheus configuration file to transfer alerts from Prometheus to Alertmanager as follows:

```

alerting:
  alertmanagers:
    - static_configs:
      - targets:
        - localhost:9093

```

In our case, we choose to send notifications via Gmail. Therefore, we configure the Gmail sender for all alert notifications in the global configuration of Alertmanager using the secure Simple Mail Transfer Protocol (SMTP) protocol:

```

global:
  smtp_from: 'alertsservice24@gmail.com'
  smtp_smarthost: smtp.gmail.com:587
  smtp_auth_username: 'alertsservice24@gmail.com'
  smtp_auth_identity: 'alertsservice24@gmail.com'
  smtp_auth_password: 'vykc auph bqmi yaru'

```

In the global section, the `smtp_from` field sets the sender's email address to `alertsservice24@gmail.com`, making this the address from which alert emails will appear to come. The `smtp_smarthost` defines the SMTP server and port used to send these emails, which is Gmail's SMTP server at `smtp.gmail.com` on port 587, used for SMTP with TLS (Transport Layer Security) for secure transmission. The `smtp_auth_username` and `smtp_auth_identity` fields both use `alertsservice24@gmail.com` as the email address for authentication purposes, ensuring that Alertmanager can log in to the SMTP server. The `smtp_auth_password` provides the password necessary for this authentication. Now, we configure the route tree in the Alertmanager configuration file to send notifications to the appropriate receivers. The route configuration is illustrated in Figure. 3.23 :

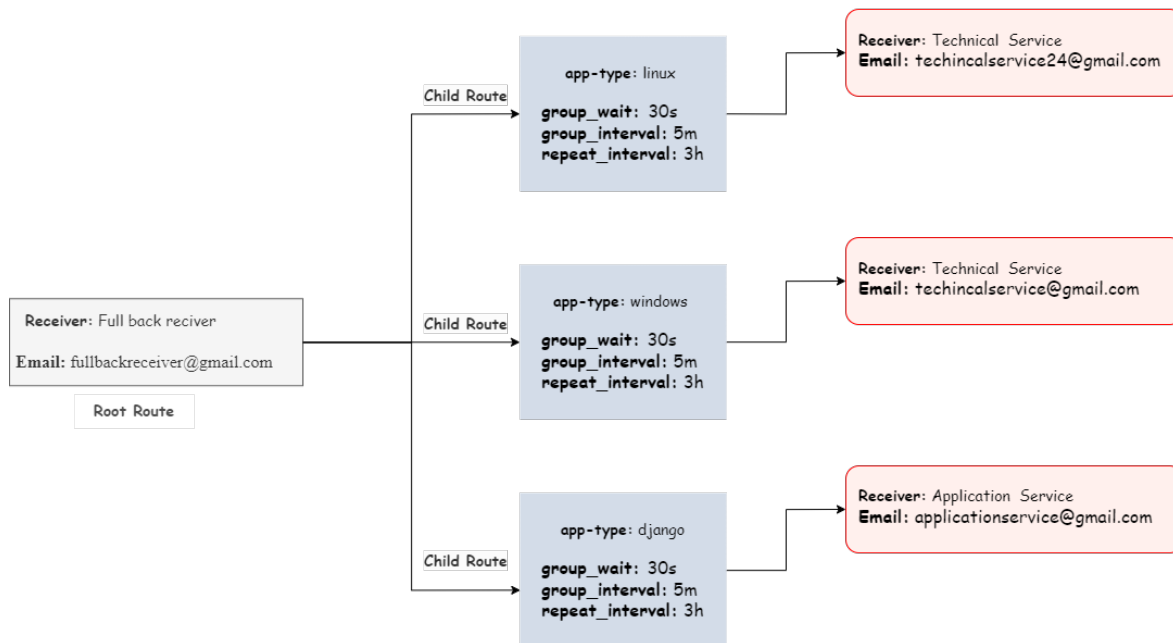


Figure 3.23: Diagram of Route Tree Implementation

As we can see, the route configuration is designed to send notifications based on the `app_type` label, which categorizes alerts into three distinct groups: Linux, Windows, and Django. Each child route within the configuration specifies the `app_type`, with parameters set for grouping and throttling alerts.

- The `group_wait` parameter is set to 30 seconds, meaning Alertmanager waits this duration before sending the first notification in a group.
- The `group_interval` is set to 5 minutes, defining how long to wait before sending a new notification if more alerts come in for the same group.

- The `repeat_interval` is set to 3 hours, specifying how often to resend the notification if the alert is still active.

Each child route directs notifications to a specific receiver based on the `app_type`: Technical Service for Linux and Windows, and Application Service for Django. Additionally, a root route is configured with a fullback receiver, which ensures that any alerts not matched by the child routes are sent to a default email address, `fullbackreceiver@gmail.com`. To reduce alert noise and prevent alert fatigue, we use inhibition to silence specific alerts when higher-priority alerts are already active. In our case, when two alerts for the same operating system are triggered: one with a critical severity and another with a warning severity, we configure Alertmanager to mute the warning severity alert if the critical severity alert is triggered. This is achieved by adding the following code to the Alertmanager configuration file:

```
inhibit_rules:  
  - source_match:  
      severity: 'critical'  
    target_match:  
      severity: 'warning'  
    equal: ['app_type', 'category']
```

This configuration ensures that when a critical severity alert is active, any corresponding warning severity alert for the same `app_type` and `category` is silenced.

3.4.5 Data Visualisation Using Grafana

After configuring Prometheus to collect target metrics, the Grafana server can be downloaded and installed to visualize the collected data.

Grafana can be accessed via a web browser at `http://localhost:3000/`. On the first visit, the user is prompted to change the login password. The default username and password are both `'admin'`.

To create dashboards for visualizing the collected data, a data source needs to be configured. Set the data source name and URL to point to the Prometheus server as shown in Figure. 3.24:

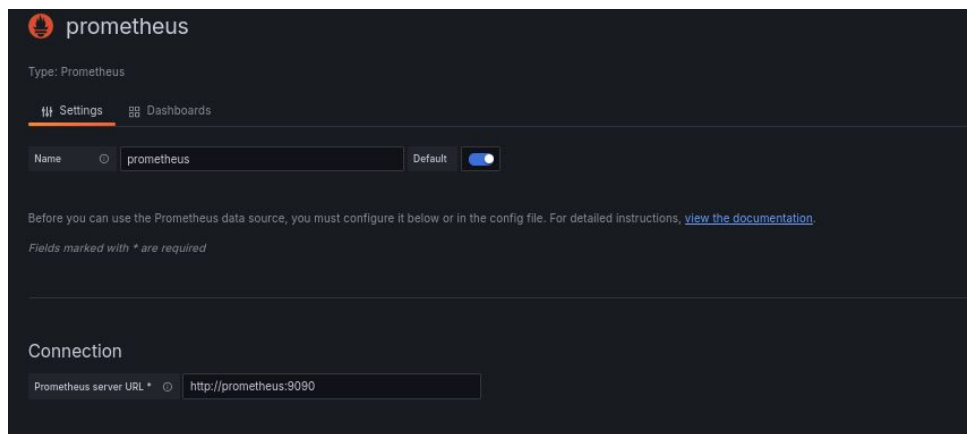


Figure 3.24: Configuring Prometheus as a Data Source for Grafana

Dashboard visualization in Grafana allows the creation of custom dashboards by configuring each panel within the target dashboard. We begin by adding a new dashboard and select Prometheus as the data source, as shown in Figure. 3.25.

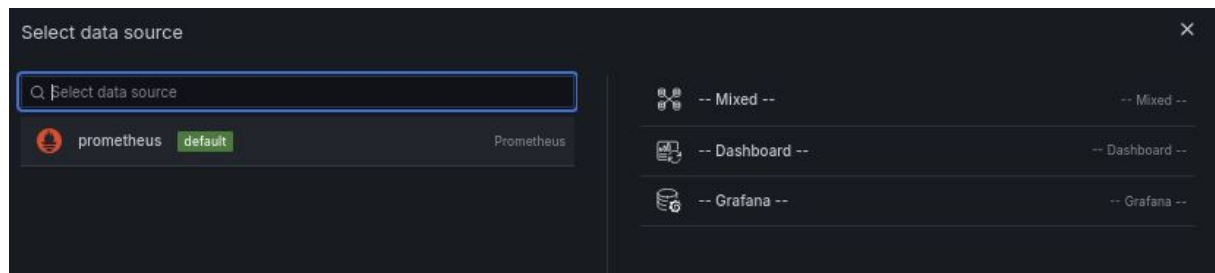


Figure 3.25: Adding Prometheus as a Data Source to a Dashboard

Creating variables in the dashboard is essential for making it dynamic and reusable. For example, to visualize the network equipments data of our architecture, we can create a variable instance to select each equipment's IP address in each panel. To create this variable, we navigate to the dashboard settings and create a new variable as shown in Figure. 3.26. We select the options "Multi-value" and "Include All" to enable multiple values to be selected simultaneously and to include an option for all variables.

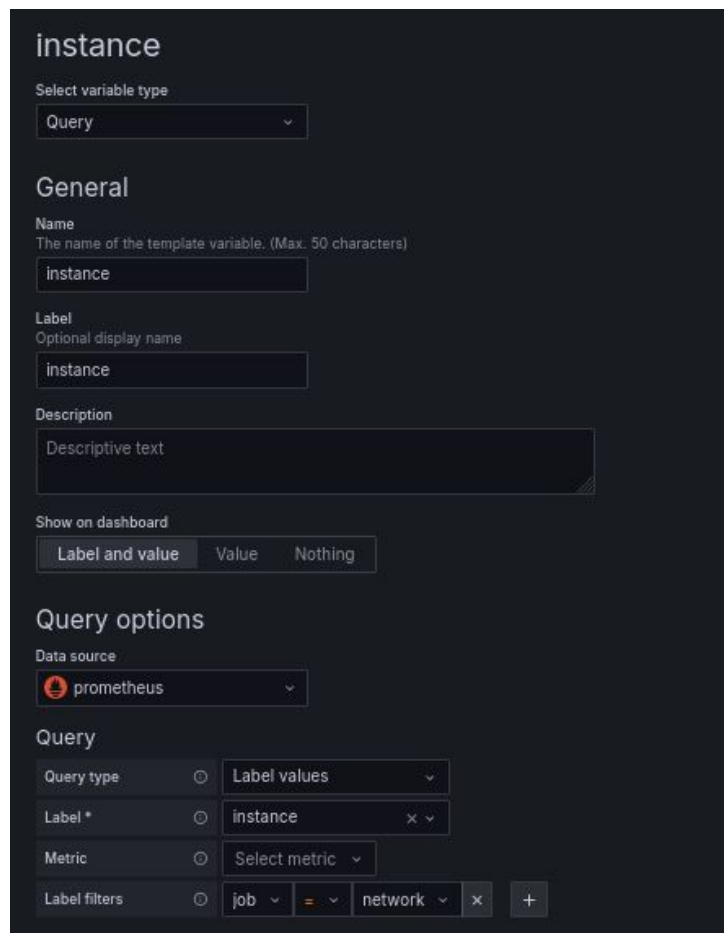


Figure 3.26: Creating an Instance Variable in Dashboard

This method is applied to all necessary variables. After creating the variables, we configure the panels with the desired metric queries using PromQL. By using the instance variable, we can create a panel to visualize the rate of incoming traffic over five minutes, configuring the query as shown in Figure Figure. 3.27.

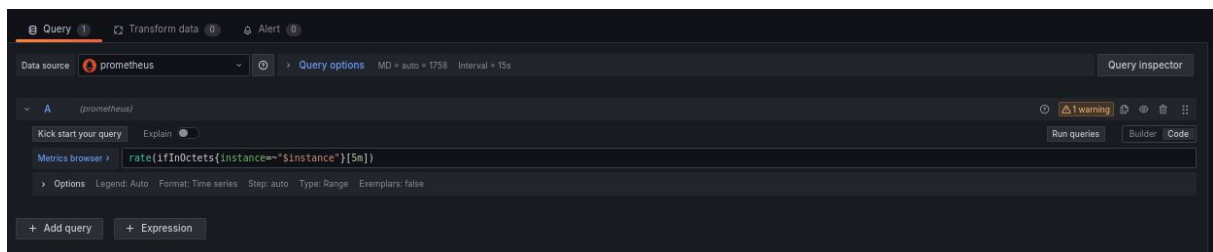


Figure 3.27: Configuring the Metric Panel

After creating the metric and adjusting the visualization options located on the right, run the query to get the metric visualization. The visualization can aggregate all equipment metrics by selecting "All" in the instance variable, or it can display metrics for a

single piece of equipment by selecting the IP address of the equipment in the instance variable, as shown in Figure. 3.29, Figure. 3.28 and Figure. 3.30.



Figure 3.28: Incoming Traffic Visualization of the Router



Figure 3.29: Incoming Traffic Visualization of the Switch



Figure 3.30: Incoming Traffic Visualization of both the Router and the Switch

In our project, we create dashboards for each target in the IT infrastructure, as shown in Figure. 3.31. We base their creation on the method we recently discussed, except for server dashboards. For servers, we utilize existing predefined dashboards for Node Exporter and WMI Exporter, which we then adjust to integrate into our project.

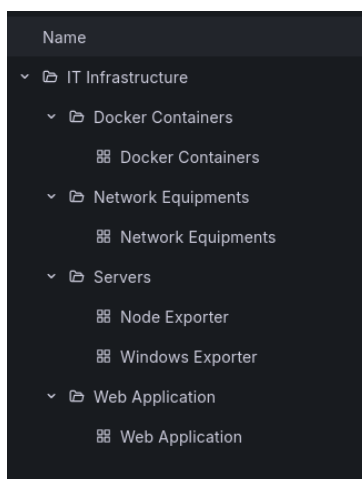


Figure 3.31: Dashboard Creation for Each Target

3.4.6 Deployment of The Monitoring System using Docker Compose

To effectively deploy and manage a comprehensive monitoring system, we have created a repository named **monitoring**. This repository serves as the main directory housing all subdirectories and the `docker-compose.yml` file for orchestrating the deployment. Within this repository, we have organized subdirectories for each primary monitoring tool: Prometheus, Alertmanager, and SNMP Exporter. Each subdirectory contains specific configuration YAML files and setup instructions tailored to the respective tool.

Our Docker Compose setup includes six containers: Prometheus, Alertmanager, SNMP Exporter, Node Exporter, cAdvisor, and Grafana, as illustrated in Figure. 3.32. This configuration enhances the flexibility of our monitoring system, allowing it to be deployed seamlessly across various environments.

In the Docker Compose YAML file, we specify the version as `version: '3.7'`. Within the **services** section, we define configurations for each service. The following snippet provides the configuration for the Prometheus service: it uses the latest `prom/prometheus` image, assigns the container name `prometheus`, and mounts local directories (`./prometheus` for configuration and `prom_data:/prometheus` for data storage). The `command` section specifies Prometheus configuration options, including the path to `prometheus.yml`, storage settings, and lifecycle management. Port 9090 is exposed for web access, and `restart: always` ensures the container automatically restarts in case of failure. Lastly, the service is connected to the monitoring network to ensure connectivity with other services within the same Docker Compose setup, as well as with the Django application service which exists in another docker compose setup.

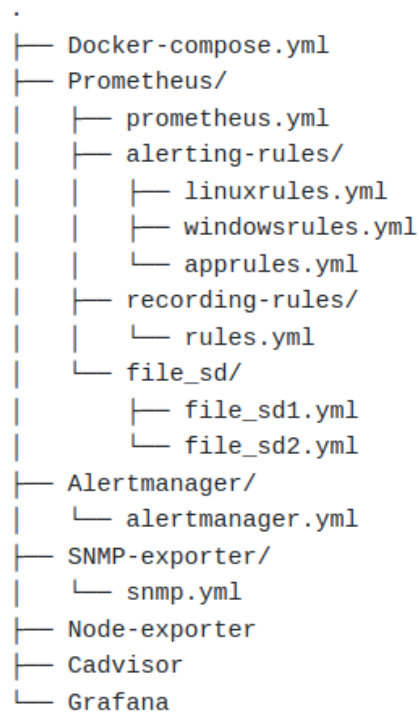


Figure 3.32: Docker Compose Services

```

version: '3.7'

services:
  prometheus:
    image: prom/prometheus:latest
    container_name: prometheus
    volumes:
      - ./prometheus:/etc/prometheus
      - prom_data:/prometheus
    command:
      - '--config.file=/etc/prometheus/prometheus.yml'
      - '--storage.tsdb.path=/prometheus'
      - '--web.enable-lifecycle'
    ports:
      - 9090:9090
    restart: always
    networks:
      - monitoring

```

```
networks:
  monitoring:
    external: true

volumes:
  prom_data:
  grafana-data:
```

The second snippet from the `docker-compose.yml` file defines crucial configurations for the monitoring setup. It includes definitions for networks and volumes necessary for the deployment. The monitoring network is specified with `external: true`, indicating its existence outside the current Docker Compose setup and ensuring connectivity with external services and applications. Volumes `prom_data` and `grafana-data` are declared to store persistent data for Prometheus and Grafana respectively, ensuring data durability and reliability even during container restarts or updates.

The full `Docker-compose.yml` file is in appendix [A.2](#).

Now, we can easily manage and deploy the entire monitoring stack with a single command.

```
$ cd monitoring
$ sudo docker-compose up -d
```

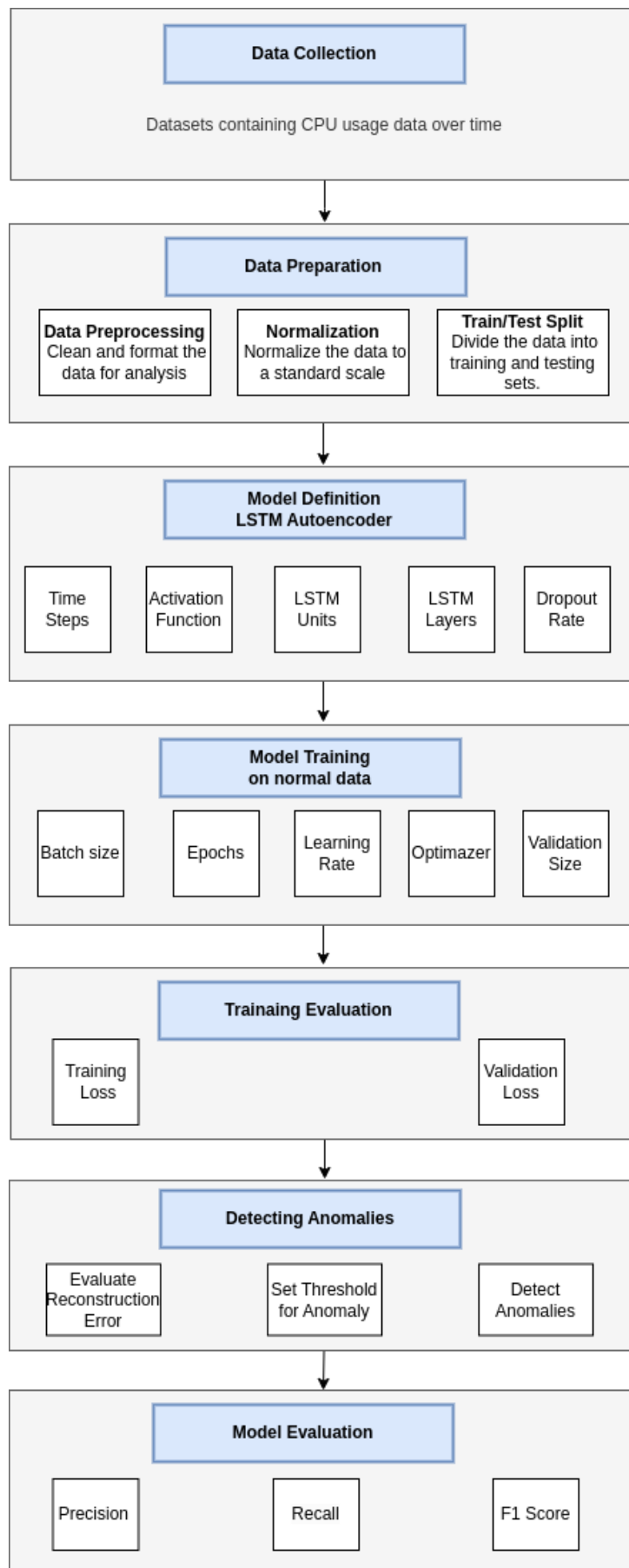
3.5 Implementation of Anomaly Detection in Time Series Data Using LSTM Autoencoders

In this section, we establish anomaly detection in time series data using deep learning LSTM autoencoders. This technique is applied to CPU usage data due to the increasing reliance on cloud infrastructure and virtual machines for storage and computing in the era of big data. High CPU usage can lead to server crashes, making early detection of anomalies essential. Effective monitoring of CPU usage helps prevent crashes and ensures smooth cloud operations.

To effectively monitor CPU usage and detect anomalies, we employ LSTM autoencoders. These deep learning models are particularly well-suited for time series data due to their ability to capture temporal dependencies. This approach has shown the best results in this field [43], [46]. By training the LSTM autoencoder on normal CPU usage patterns,

we can identify deviations that indicate potential anomalies.

Diagram 3.33 shows the steps we carried out to implement and utilize LSTM autoencoders for anomaly detection in CPU usage. The process begins with data collection and preprocessing, followed by the training of the LSTM autoencoder on normal CPU usage patterns. Once trained, the model is used to monitor CPU data and detect anomalies.



65
Figure 3.33: Explanatory Diagram of the Steps Carried Out

Let's explain each step in detail.

3.5.0.1 Data Collection

The chosen dataset is sourced from Amazon SageMaker [47] and includes timestamps and CPU values ranging from 0 to 100. It comprises 18,051 samples, spanning from May 14, 2014, at 01:14:00 to July 15, 2014, at 17:19:00, with timestamps recorded every 5 minutes. As shown in Figure. 3.34. This dataset contains two anomalies at '2014-07-12 02:04:00' and '2014-07-14 21:44:00'.

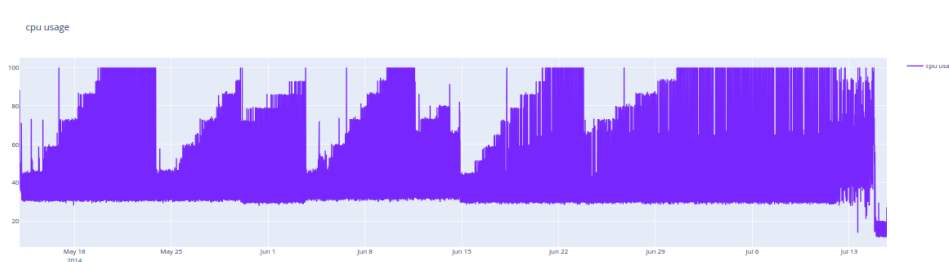


Figure 3.34: CPU Usage Dataset

3.5.0.2 Data Preparation

Before using the dataset for training or testing, thorough preparation is essential. The first step is data preprocessing, which involves cleaning the data and handling any missing values by imputation. Next, we apply scaling to the data to standardize the features. Finally, the dataset is split into training and testing sets. Typically, 90% of the data (comprising normal instances without anomalies) is allocated for training, while the remaining 10% is used for testing. Anomalies are included only in the testing dataset for evaluation purposes.

3.5.0.3 Model Definition

Before presenting the experimental results, let's define the LSTM model configuration. The model is characterized by several key parameters: the number of LSTM layers stacked, the number of units (neurons) in each layer, the activation function used, and the dropout rate. Each parameter configuration influences the model's ability to learn and generalize from the data. The description of each parameter is given in table 3.2.

Parameter	Description
LSTM Layers	Number of LSTM layers stacked in the model
LSTM Units	Number of units (neurons) in each LSTM layer
Time Steps	Number of time steps or sequence length for each input sample
Activation Function	Activation function used in LSTM layers (e.g., sigmoid, tanh, relu)
Dropout Rate	Fraction of neurons to randomly drop during training

Table 3.2: Parameters in an LSTM model

Below in table 3.3, we present the optimal parameter values that yielded the best results. These values were determined by varying the parameters to assess their impact on the model’s performance.

Parameter	Optimal Value
LSTM Layers	2
LSTM Units	128 in layer1/ 64 in layer 2
Time Steps	10 samples
Activation Function	relu
Dropout Rate	0.2

Table 3.3: Optimal parameter values for the LSTM model

3.5.0.4 Model Training

In the model training step, we configure several key parameters to optimize the performance of the LSTM autoencoder. The batch size determines the number of samples processed before the model’s internal parameters are updated. Epochs refer to the number of complete passes through the entire training dataset. The learning rate controls the size of the steps the optimizer takes to minimize the loss function. The optimizer adjusts the learning process to improve convergence. Additionally, a portion of the data is set aside as the validation set to monitor the model’s performance and prevent overfitting.

Below is a table summarizing these parameters and their descriptions:

Parameter	Description
Batch Size	Number of samples processed before updating model parameters
Epochs	Number of complete passes through the entire training dataset
Learning Rate	Step size used by the optimizer to update model parameters
Optimizer	Algorithm used to adjust the learning process (e.g., Adam, RMSprop)
Validation Size	Proportion of data set aside to evaluate model performance during training

Table 3.4: Training parameters for the LSTM autoencoder

Based on our experiments, we identified the optimal training parameters for the LSTM

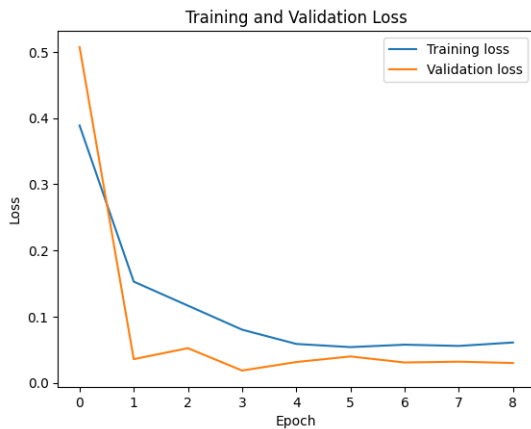
autoencoder. We found that a batch size of 32, running for 9 epochs, with a learning rate of 0.01, and using the Adam optimizer yielded the best results. Additionally, a validation size of 10% was used to monitor the model’s performance and prevent overfitting during training. Below is a table summarizing these optimal values:

Parameter	Optimal Value
Batch Size	32
Epochs	9
Learning Rate	0.01
Optimizer	Adam
Validation Size	10%

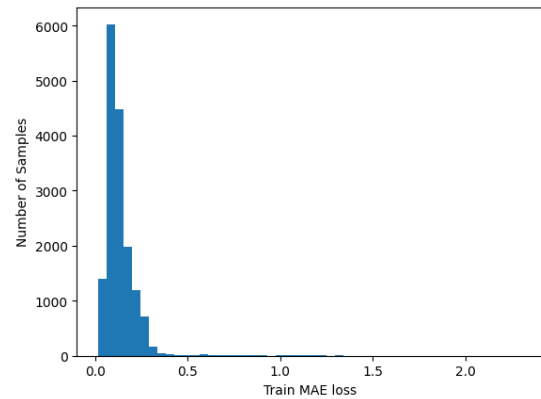
Table 3.5: Optimal training parameters for the LSTM autoencoder

3.5.0.5 Training Evaluation

In the training evaluation step, we assess the model’s performance using Mean Squared Error (MSE) loss for both training and validation datasets. Below in Figure. 3.35a is the graph of training and validation loss in terms of the number of epochs:



(a) Training and Validation MSE Loss over Epochs



(b) Train MAE Loss Histogram

Figure 3.35: Evaluation of Training Process

The graph in Figure. 3.35a shows the training and validation MSE loss over 9 epochs. Both losses start high but decrease significantly within the first few epochs. By the 3rd epoch, the validation loss stabilizes, indicating that the model is generalizing well. The training loss continues to decrease slightly, suggesting the model is fine-tuning without overfitting.

The histogram in Figure. 3.35b illustrates the distribution of Mean Absolute Error (MAE) loss for the training samples. The majority of the samples have a very low MAE

loss, concentrated around 0.0 to 0.1, indicating that the model predictions are close to the actual values for most of the training data. There are a few samples with higher MAE losses, but they are relatively infrequent.

3.5.0.6 Detecting Anomalies

In the anomaly detection step, we calculate the reconstruction error for each data point. The reconstruction error is the difference between the original data and the data reconstructed by the LSTM autoencoder. To identify anomalies, we set a threshold value. Based on various experiments, the following threshold has provided the best results. `THRESHOLD = np.max(reconstruction_error) * 0.78` This threshold is determined by taking 78% of the maximum reconstruction error observed during the training phase, calculated as:

Data points with reconstruction errors exceeding this threshold are flagged as anomalies.

3.6 Conclusion

In conclusion, this chapter has detailed the design and implementation of our project's monitoring system. By deploying Prometheus for metric collection, integrating exporters for targeted metrics, and utilizing Grafana for data visualization, we have established a robust framework for real-time monitoring. Moreover, our exploration of advanced anomaly detection techniques, such as LSTM autoencoders, demonstrates a proactive approach to ensuring system reliability.

Chapter 4

Results and Validation

4.1 Introduction

In this chapter, we present the results and validation of the monitoring system setup. This includes a detailed overview of the performance and status of the entire network architecture, demonstrating how effectively Prometheus and Grafana can track and visualize metrics across different technologies. Additionally, we evaluate the anomaly detection model using LSTM autoencoders, specifically focusing on CPU usage. The evaluation is conducted using appropriate parameters to ensure the precision and reliability of the anomaly detection mechanism.

4.2 Validation of the Monitoring system

In this section, we present the results and dashboards from the monitoring system setup. These dashboards illustrate the performance and status of the entire network architecture, showcasing how effectively Prometheus and Grafana can track and visualize metrics across different technologies.

4.2.1 Prometheus

After configuring Prometheus to scrape metrics from targets, it is essential to validate that metrics are being successfully collected. This validation can be performed on the Targets page in Prometheus. This page lists all the scrape targets and their current status, as shown in [Figure. 4.1](#) and [Figure. 4.2](#) and [Figure. 4.3](#).

Targets

All scrape pools ▾ **All** Unhealthy Collapse All

Grafana (1/1 up) [show less](#)

Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://grafana:3000/metrics	UP	instance="grafana:3000" job="Grafana" ▾	1.909s ago	44.501ms	

Prometheus (1/1 up) [show less](#)

Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://prometheus:9090/metrics	UP	instance="prometheus:9090" job="Prometheus" ▾	4.619s ago	46.832ms	

VM-Kali (1/1 up) [show less](#)

Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://node-exporter:9100/metrics	UP	instance="node-exporter:9100" job="VM-Kali" os="Kali GNU/Linux Rolling" ▾	8.998s ago	85.017ms	

Figure 4.1: Scrape Targets Listed on Prometheus Targets Page

VM-Ubuntu (1/1 up) [show less](#)

Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://192.168.56.112:9100/metrics	UP	instance="192.168.56.112:9100" job="VM-Ubuntu" ▾	8.668s ago	91.596ms	

VM-Windows10 (1/1 up) [show less](#)

Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://192.168.56.200:9182/metrics	UP	instance="192.168.56.200:9182" job="VM-Windows10" ▾	17.411s ago	3.81s	

cadvisor (1/1 up) [show less](#)

Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://cadvisor:8080/metrics	UP	instance="cadvisor:8080" job="cadvisor" ▾	6.575s ago	403.846ms	

Figure 4.2: Continuation 1 of Scraping Targets Listed on the Prometheus Targets Page

django-app (1/1 up) [show less](#)

Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://web:8000/metrics	UP	instance="web:8000" job="django-app" ▾	10.643s ago	21.518ms	

network (2/2 up) [show less](#)

Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://snmp-exporter:9116/snmp auth="router" module="standard_mibs" target="192.168.56.111"	UP	instance="192.168.56.111" job="network" ▾	1.696s ago	425.579ms	
http://snmp-exporter:9116/snmp auth="switch" module="standard_mibs" target="192.168.56.113"	UP	instance="192.168.56.113" job="network" ▾	9.803s ago	3.167s	

snmp-exporter (1/1 up) [show less](#)

Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://snmp-exporter:9116/metrics	UP	instance="snmp-exporter:9116" job="snmp-exporter" ▾	8.113s ago	11.686ms	

Figure 4.3: Continuation 2 of Scraping Targets Listed on the Prometheus Targets Page

As depicted in the figure, all configured targets from the prometheus.yml file are displayed on the Targets page. Each target’s status is shown as 'UP' along with the last scrape time and scrape duration. This information confirms that Prometheus is successfully scraping metrics from the targets.

We can also check the recording rules on the Rules page in Prometheus to ensure they are configured correctly, Figures Figure. 4.4, Figure. 4.5 and Figure. 4.6 provide confirmation.

DjangoApp

Rule	State	Error	Last Evaluation	Evaluation Time
record: job:django_http_requests_rate:5m expr: rate(django_http_requests_total_by_method_total{instance=~"\$instance",job="django-app"})[5m]	OK		13.358s ago	0.504ms
record: job:django_memory_usage:percentage expr: (sum(process_resident_memory_bytes{instance=~"\$instance",job="django-app"}) / sum(node_memory_MemTotal_bytes{job="VM-Kali"})) * 100	OK		13.358s ago	0.572ms

Figure 4.4: Validation of Django App Recording Rules on the Prometheus Rules Page

Linux

Rule	State	Error	Last Evaluation	Evaluation Time
record: job:node_cpu_usage:percentage expr: 100 * (1 - avg by (instance) (rate(node_cpu_seconds_total{mode="idle"}[5m])))	OK		9.982s ago	0.698ms
record: job:node_memory_usage:percentage expr: 100 * (1 - avg by (instance) ((node_memory_MemFree_bytes / node_memory_MemTotal_bytes)))	OK		9.983s ago	0.400ms

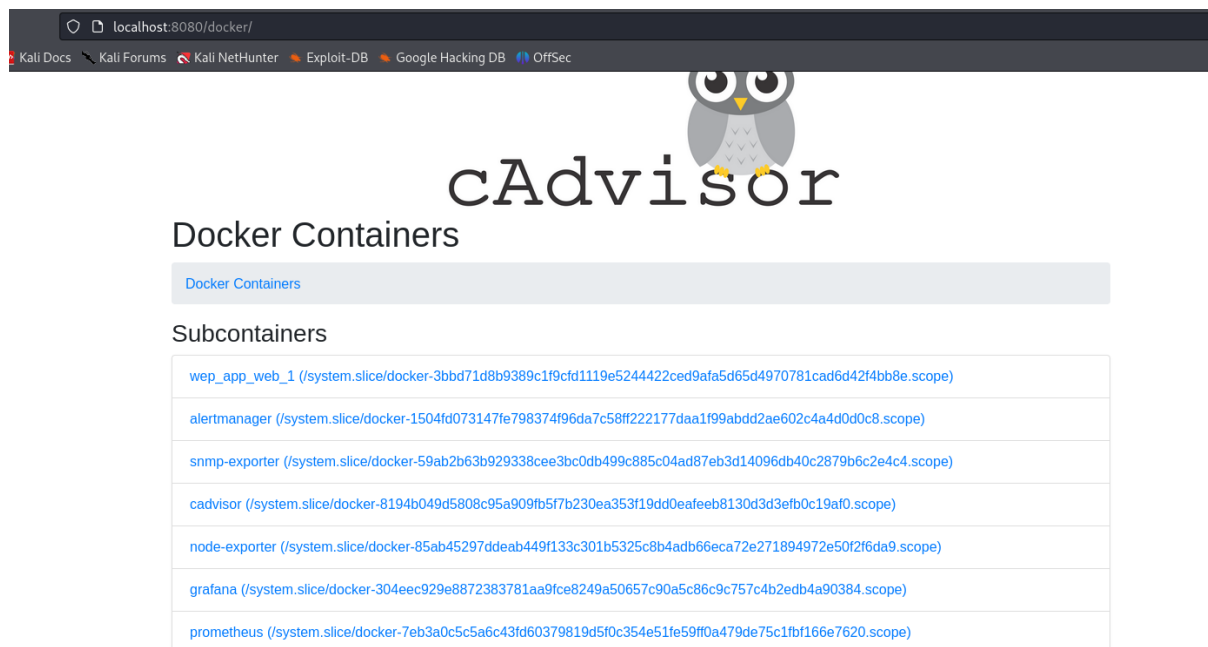
Figure 4.5: Validation of Linux Recording Rules on the Prometheus Rules Page

Windows		Interval: 15.0s	7.43s ago	1.117ms	
Rule		State	Error	Last Evaluation	Evaluation Time
record: <code>job:windows_cpu_usage:percentage</code> expr: <code>100 * avg without (cpu) (sum by (instance) (irate(windows_cpu_time_total{mode!=""}[5m])))</code>		OK		7.43s ago	0.747ms
record: <code>job:windows_memory_usage:percentage</code> expr: <code>100 * (1 - avg by (instance) ((windows_os_physical_memory_free_bytes / windows_cs_physical_memory_bytes)))</code>		OK		7.43s ago	0.336ms

Figure 4.6: Validation of Windows Recording Rules on the Prometheus Rules Page

4.2.2 Container Status with CAdvisor

cAdvisor is up and accessible via port 8080. As shown in Figure. 4.7, the seven containers, including the Django web app, are visible.



The screenshot shows the cAdvisor web interface in a browser window. The URL is localhost:8080/docker/. The page title is "cAdvisor" with a logo of an owl. Below the title, it says "Docker Containers". There is a "Docker Containers" button. Underneath, it says "Subcontainers" and lists seven containers with their IDs and scopes:

- wep_app_web_1 (/system.slice/docker-3bbd71d8b9389c1f9cfd1119e5244422ced9afa5d65d4970781cad6d42f4bb8e.scope)
- alertmanager (/system.slice/docker-1504fd073147fe798374f96da7c58ff222177daa1f99abdd2ae602c4a4d0d0c8.scope)
- snmp-exporter (/system.slice/docker-59ab2b63b929338cee3bc0db499c885c04ad87eb3d14096db40c2879b6c2e4c4.scope)
- cadvisor (/system.slice/docker-8194b049d5808c95a909fb5f7b230ea353f19dd0eafeeb8130d3d3efb0c19af0.scope)
- node-exporter (/system.slice/docker-85ab45297ddeab449f133c301b5325c8b4adb66eca72e271894972e50f2f6da9.scope)
- grafana (/system.slice/docker-304eec929e8872383781aa9fce8249a50657c90a5c86c9c757c4b2edb4a90384.scope)
- prometheus (/system.slice/docker-7eb3a0c5c5a6c43fd60379819d5f0c354e51fe59ff0a479de75c1fbf166e7620.scope)

Figure 4.7: Monitoring Containers using Cadvisor

This interface allows us to monitor the resource usage and performance of each container in real time. Figure. 4.8 and Figure. 4.9 show the real time monitoring of the web app.

4.2.3 Exporters

- **Node Exporter and WMI Exporter**

Node Exporter and WMI Exporter are operational and exposing the collected metrics from the servers to HTTP endpoints, as illustrated in Figures Figure. 4.10 and Figure. 4.11.

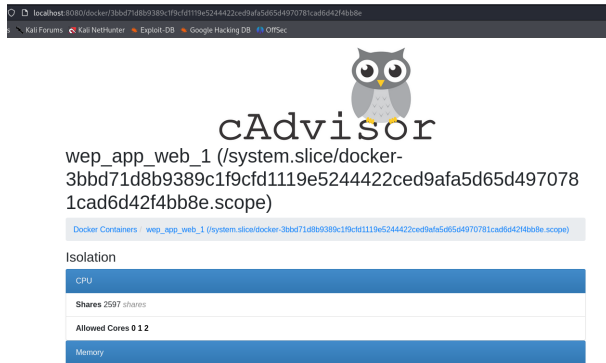


Figure 4.8: Monitoring the Web App Container using Cadvisor



Figure 4.9: Network Throughput and Errors Monitoring with cAdvisor for the Web App

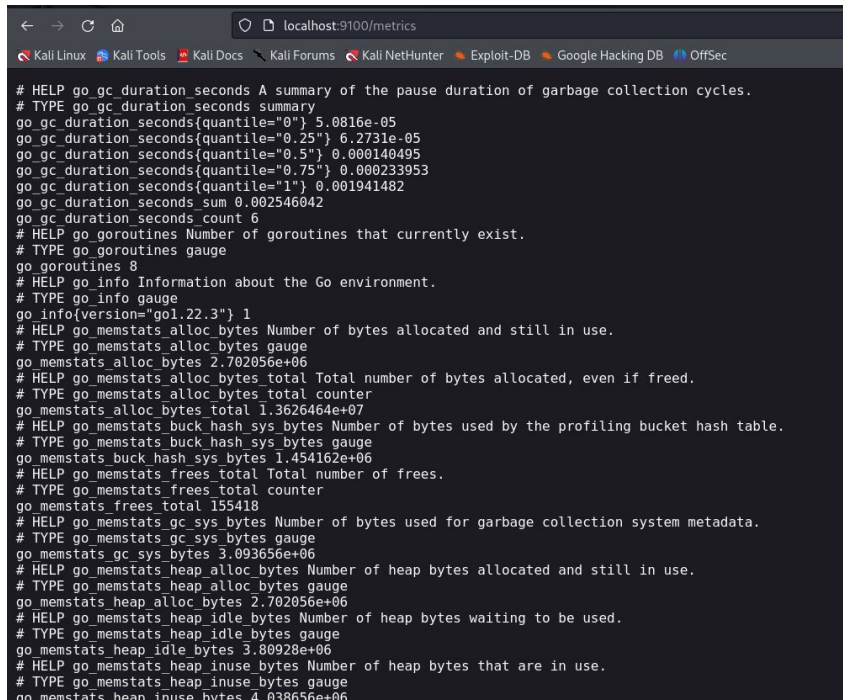


Figure 4.10: VM-Kali Metrics Exposed by Node Exporter

```

# HELP go_gc_duration_seconds A summary of the pause duration of garbage collection cycles.
# TYPE go_gc_duration_seconds summary
go_gc_duration_seconds{quantile="0"} 0
go_gc_duration_seconds{quantile="0.25"} 0
go_gc_duration_seconds{quantile="0.5"} 0
go_gc_duration_seconds{quantile="0.75"} 0
go_gc_duration_seconds{quantile="1"} 0
go_gc_duration_seconds_sum 0
go_gc_duration_seconds_count 3
# HELP go_goroutines Number of goroutines that currently exist.
# TYPE go_goroutines gauge
go_goroutines 31
# HELP go_info Information about the Go environment.
# TYPE go_info gauge
go_info{version="go1.21.9"} 1
# HELP go_memstats_alloc_bytes Number of bytes allocated and still in use.
# TYPE go_memstats_alloc_bytes gauge
go_memstats_alloc_bytes 7.958744e+06
# HELP go_memstats_alloc_bytes_total Total number of bytes allocated, even if freed.
# TYPE go_memstats_alloc_bytes_total counter
go_memstats_alloc_bytes_total 1.3364128e+07
# HELP go_memstats_buck_hash_sys_bytes Number of bytes used by the profiling bucket hash table.
# TYPE go_memstats_buck_hash_sys_bytes gauge
go_memstats_buck_hash_sys_bytes 1.452101e+06
# HELP go_memstats_frees_total Total number of frees.
# TYPE go_memstats_frees_total counter
go_memstats_frees_total 81175
# HELP go_memstats_gc_sys_bytes Number of bytes used for garbage collection system metadata.
# TYPE go_memstats_gc_sys_bytes gauge
go_memstats_gc_sys_bytes 2.997192e+06
# HELP go_memstats_heap_alloc_bytes Number of heap bytes allocated and still in use.
# TYPE go_memstats_heap_alloc_bytes gauge
go_memstats_heap_alloc_bytes 7.958744e+06
# HELP go_memstats_heap_idle_bytes Number of heap bytes waiting to be used

```

Figure 4.11: VM-Windows Metrics Exposed by WMI Exporter

- **SNMP Exporter**

The SNMP exporter is up and running, accessible via the Kali IP address and port 9116. In Figure. 4.12, we see the interface of the SNMP exporter. The interface includes three fields: target, authentication, and module. We fill these fields with the router information, and upon clicking 'submit', we can visualize the router metrics as shown in Figure. 4.13.

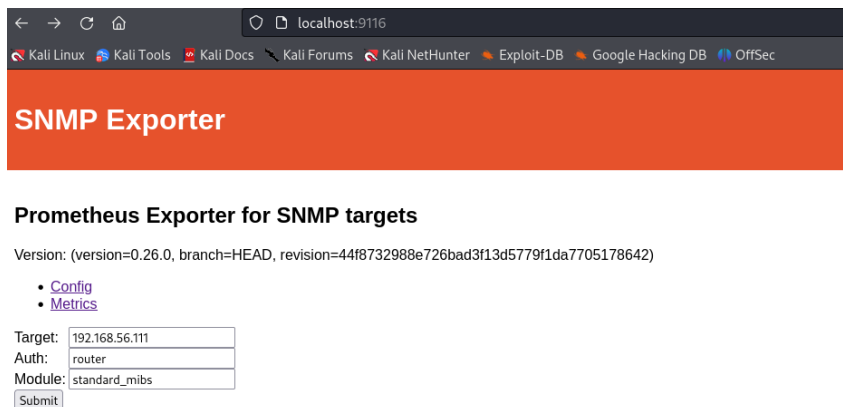


Figure 4.12: SNMP Exporter User Interface


```

# HELP ifInOctets Incoming octets on the interface
# TYPE ifInOctets counter
ifInOctets{ifDescr="",ifIndex="1"} 1.847304e+06
ifInOctets{ifDescr="",ifIndex="2"} 1.806534e+06
ifInOctets{ifDescr="",ifIndex="3"} 0
ifInOctets{ifDescr="",ifIndex="4"} 0
ifInOctets{ifDescr="",ifIndex="5"} 0
ifInOctets{ifDescr="",ifIndex="6"} 0
# HELP ifIndex
# TYPE ifIndex gauge
ifIndex{ifAlias="",ifDescr="FastEthernet0/0",ifIndex="1",ifName="Fa0/0"} 1
ifIndex{ifAlias="",ifDescr="GigabitEthernet1/0",ifIndex="2",ifName="Gi1/0"} 2
ifIndex{ifAlias="",ifDescr="GigabitEthernet2/0",ifIndex="3",ifName="Gi2/0"} 3
ifIndex{ifAlias="",ifDescr="NVI0",ifIndex="6",ifName="NV0"} 6
ifIndex{ifAlias="",ifDescr="Null0",ifIndex="5",ifName="Nu0"} 5
ifIndex{ifAlias="",ifDescr="VoIP-Null0",ifIndex="4",ifName="Vo0"} 4
# HELP ifOperStatus Interface operational status
# TYPE ifOperStatus gauge
ifOperStatus{ifIndex="1"} 1
ifOperStatus{ifIndex="2"} 1
ifOperStatus{ifIndex="3"} 2
ifOperStatus{ifIndex="4"} 1
ifOperStatus{ifIndex="5"} 1
ifOperStatus{ifIndex="6"} 1
# HELP ifOutOctets outgoing octets on the interface
# TYPE ifOutOctets counter
ifOutOctets{ifDescr="",ifIndex="1"} 4.295291e+06
ifOutOctets{ifDescr="",ifIndex="2"} 772287
ifOutOctets{ifDescr="",ifIndex="3"} 0
ifOutOctets{ifDescr="",ifIndex="4"} 0
ifOutOctets{ifDescr="",ifIndex="5"} 0
ifOutOctets{ifDescr="",ifIndex="6"} 0
# HELP ifSpeed
# TYPE ifSpeed gauge
ifSpeed{ifDescr="FastEthernet0/0",ifIndex="1",ifName="Fa0/0"} 1e+08
ifSpeed{ifDescr="GigabitEthernet1/0",ifIndex="2",ifName="Gi1/0"} 1e+09
ifSpeed{ifDescr="GigabitEthernet2/0",ifIndex="3",ifName="Gi2/0"} 1e+09
ifSpeed{ifDescr="NVI0",ifIndex="6",ifName="NV0"} 56000
ifSpeed{ifDescr="Null0",ifIndex="5",ifName="Nu0"} 4.294967295e+09
ifSpeed{ifDescr="VoIP-Null0",ifIndex="4",ifName="Vo0"} 4.294967295e+09
# HELP snmp_scrape_duration_seconds Total SNMP time scrape took (walk and processing).
# TYPE snmp_scrape_duration_seconds gauge
snmp_scrape_duration_seconds{module="standard mibs"} 0.17503714
# HELP snmp_scrape_packets_retried Packets retried for get, bulkget, and walk.
# TYPE snmp_scrape_packets_retried gauge
snmp_scrape_packets_retried{module="standard mibs"} 0

```

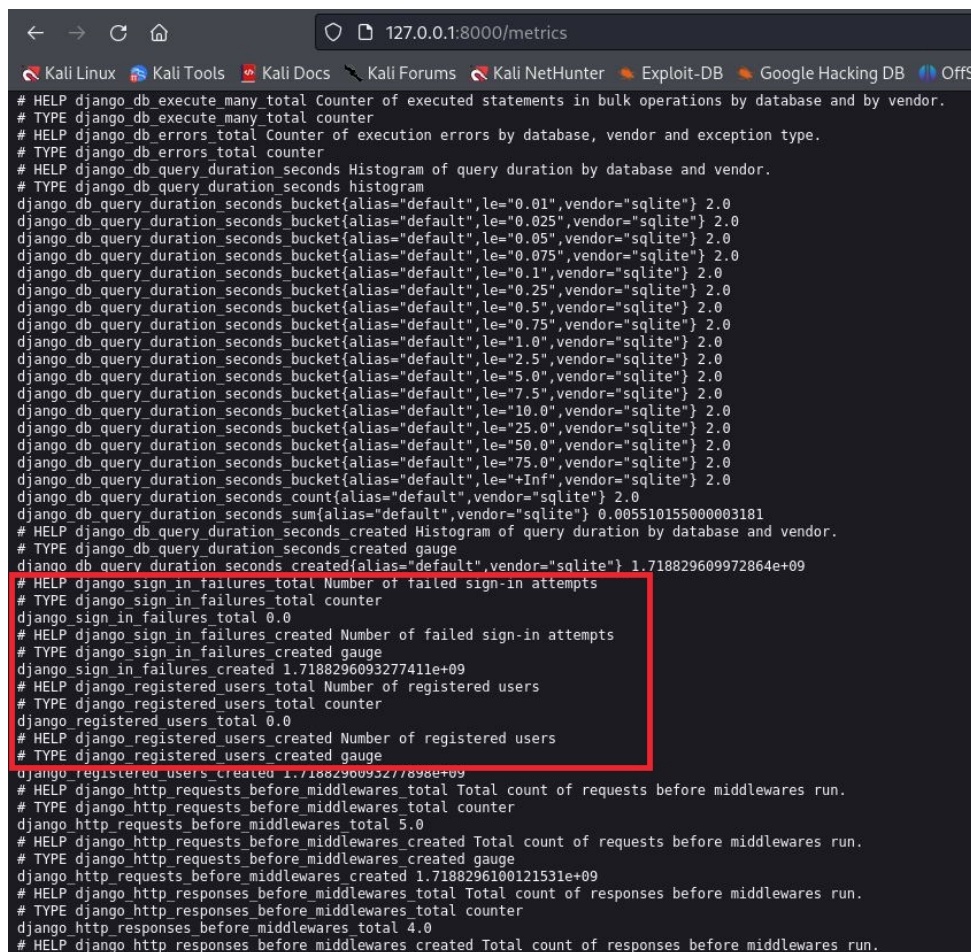
Figure 4.13: Router Metrics in SNMP Exporter HTTP Endpoint

The metrics shown in the red frame in Figure 4.13 represent the operational status of each router interface. For instance, the GigabitEthernet2/0 interface was intentionally shut down, changing its ifOperStatus metric to 2, indicating the interface is down. The meanings of each number in ifOperStatus are as follows:

- 1: Up - The interface is operational.
- 2: Down - The interface is not operational.
- 3: Testing - The interface is in testing mode.
- 4: Unknown - The operational status of the interface is unknown.
- 5: Dormant - The interface is waiting for an external event.
- 6: NotPresent - Some component (typically, a hardware component) is missing.
- 7: LowerLayerDown - The interface itself is OK, but an interface it depends on is down.

4.2.4 Web Application

To verify the instrumentation of the web application, we access the `/metrics` endpoint in the web browser using `http://127.0.0.1/metrics`, as depicted in Figure 4.14. Additionally, to ensure that these metrics are scraped by Prometheus for monitoring, we filter the metrics by the job name `'django-app'`, which is configured in `prometheus.yml`, as shown in Figure 4.15.



```

127.0.0.1:8000/metrics
# HELP django_db_execute_many_total Counter of executed statements in bulk operations by database and by vendor.
# TYPE django_db_execute_many_total counter
# HELP django_db_errors_total Counter of execution errors by database, vendor and exception type.
# TYPE django_db_errors_total counter
# HELP django_db_query_duration_seconds Histogram of query duration by database and vendor.
# TYPE django_db_query_duration_seconds histogram
django_db_query_duration_seconds_bucket{alias="default",le="0.01",vendor="sqlite"} 2.0
django_db_query_duration_seconds_bucket{alias="default",le="0.025",vendor="sqlite"} 2.0
django_db_query_duration_seconds_bucket{alias="default",le="0.05",vendor="sqlite"} 2.0
django_db_query_duration_seconds_bucket{alias="default",le="0.075",vendor="sqlite"} 2.0
django_db_query_duration_seconds_bucket{alias="default",le="0.1",vendor="sqlite"} 2.0
django_db_query_duration_seconds_bucket{alias="default",le="0.25",vendor="sqlite"} 2.0
django_db_query_duration_seconds_bucket{alias="default",le="0.5",vendor="sqlite"} 2.0
django_db_query_duration_seconds_bucket{alias="default",le="0.75",vendor="sqlite"} 2.0
django_db_query_duration_seconds_bucket{alias="default",le="1.0",vendor="sqlite"} 2.0
django_db_query_duration_seconds_bucket{alias="default",le="2.5",vendor="sqlite"} 2.0
django_db_query_duration_seconds_bucket{alias="default",le="5.0",vendor="sqlite"} 2.0
django_db_query_duration_seconds_bucket{alias="default",le="7.5",vendor="sqlite"} 2.0
django_db_query_duration_seconds_bucket{alias="default",le="10.0",vendor="sqlite"} 2.0
django_db_query_duration_seconds_bucket{alias="default",le="25.0",vendor="sqlite"} 2.0
django_db_query_duration_seconds_bucket{alias="default",le="50.0",vendor="sqlite"} 2.0
django_db_query_duration_seconds_bucket{alias="default",le="75.0",vendor="sqlite"} 2.0
django_db_query_duration_seconds_bucket{alias="default",le="+Inf",vendor="sqlite"} 2.0
django_db_query_duration_seconds_count{alias="default",vendor="sqlite"} 2.0
django_db_query_duration_seconds_sum{alias="default",vendor="sqlite"} 0.005510155000003181
# HELP django_db_query_duration_seconds_created Histogram of query duration by database and vendor.
# TYPE django_db_query_duration_seconds_created gauge
django_db_query_duration_seconds_created{alias="default",vendor="sqlite"} 1.718829609972864e+09
# HELP django_sign_in_failures_total Number of failed sign-in attempts
# TYPE django_sign_in_failures_total counter
django_sign_in_failures_total 0.0
# HELP django_sign_in_failures_created Number of failed sign-in attempts
# TYPE django_sign_in_failures_created gauge
django_sign_in_failures_created 1.7188296093277411e+09
# HELP django_registered_users_total Number of registered users
# TYPE django_registered_users_total counter
django_registered_users_total 0.0
# HELP django_registered_users_created Number of registered users
# TYPE django_registered_users_created gauge
django_registered_users_created 1.7188296093277411e+09
# HELP django_http_requests_before_middleware_total Total count of requests before middlewares run.
# TYPE django_http_requests_before_middleware_total counter
django_http_requests_before_middleware_total 5.0
# HELP django_http_requests_before_middleware_created Total count of requests before middlewares run.
# TYPE django_http_requests_before_middleware_created gauge
django_http_requests_before_middleware_created 1.7188296100121531e+09
# HELP django_http_responses_before_middleware_total Total count of responses before middlewares run.
# TYPE django_http_responses_before_middleware_total counter
django_http_responses_before_middleware_total 4.0
# HELP django_http_responses_before_middleware_created Total count of responses before middlewares run.

```

Figure 4.14: Web Application Metrics Exposed on Metrics Endpoint

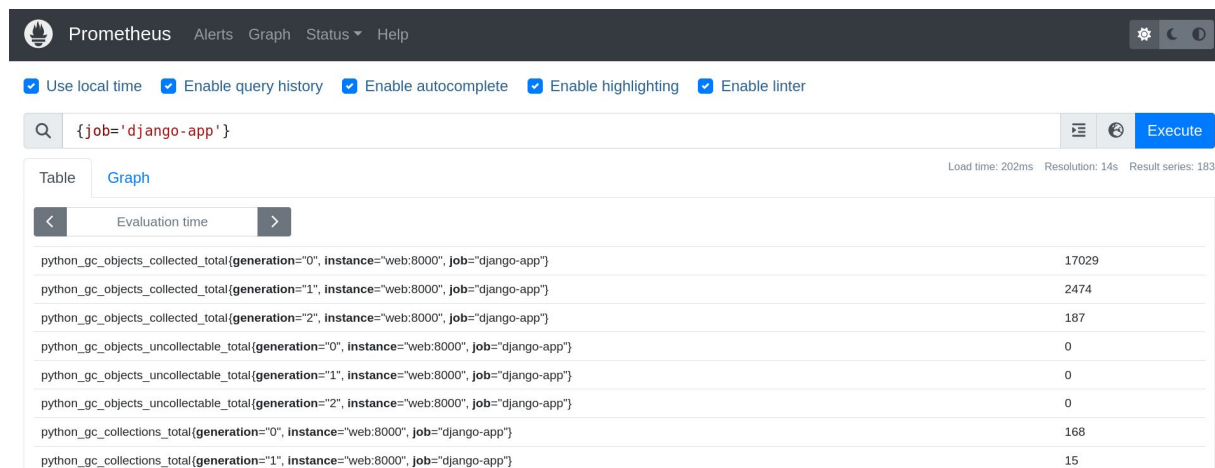


Figure 4.15: Web Application Metrics scraped by Prometheus

As we can see the two figures demonstrate the successful instrumentation of a Django web application using both the `django_prometheus` library and the Prometheus client library. The metrics shown in the red frame in Figure. 4.14 are the custom metrics created using the Prometheus client library, while the others are generated by the `django_prometheus` library.

4.2.5 Alerting

Once we run Alertmanager and restart Prometheus, the Prometheus Alerts interface loads alerts from the configuration file, as shown in Figure. 4.16:

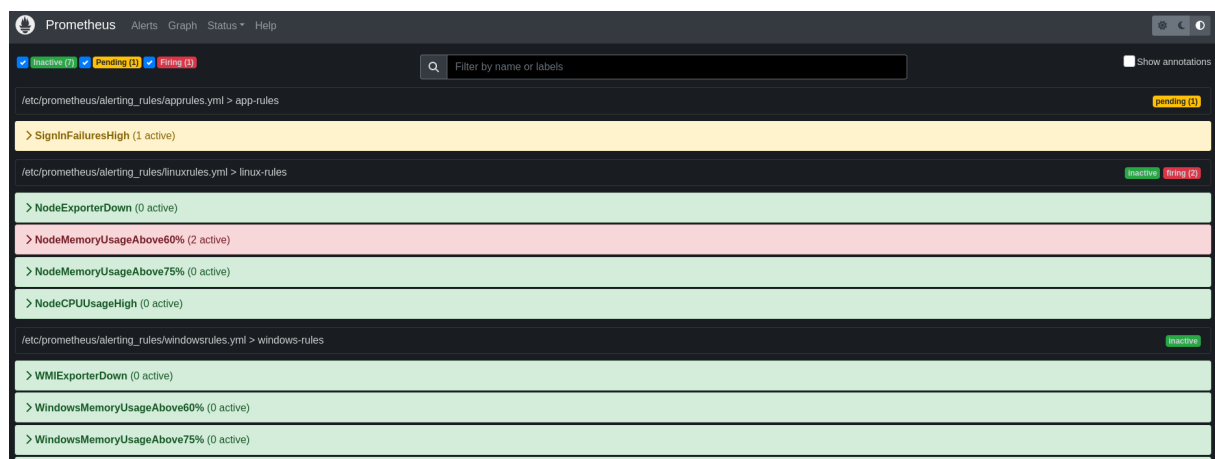


Figure 4.16: Different Alerts Displayed in Prometheus Alerts Interface

This interface represents the status of various alerts configured in Prometheus. Each alert's status (inactive, pending, or firing) helps operators quickly identify issues:

- Green alerts refer to inactive alerts, indicating that the monitored conditions are normal.
- Yellow alerts refer to pending alerts. In this case, we have one pending alert named `SignFailureHigh`, which indicates that the condition has been detected but is waiting for the specified duration before firing the alert (we have already configured it for 2 minutes).
- Red alerts refer to firing alerts. In this instance, there is one firing alert indicating that the condition has persisted for the defined duration, and the alert is actively being triggered.

Alertmanager displays the fired alerts within its web interface, as shown in Figure. 4.17.

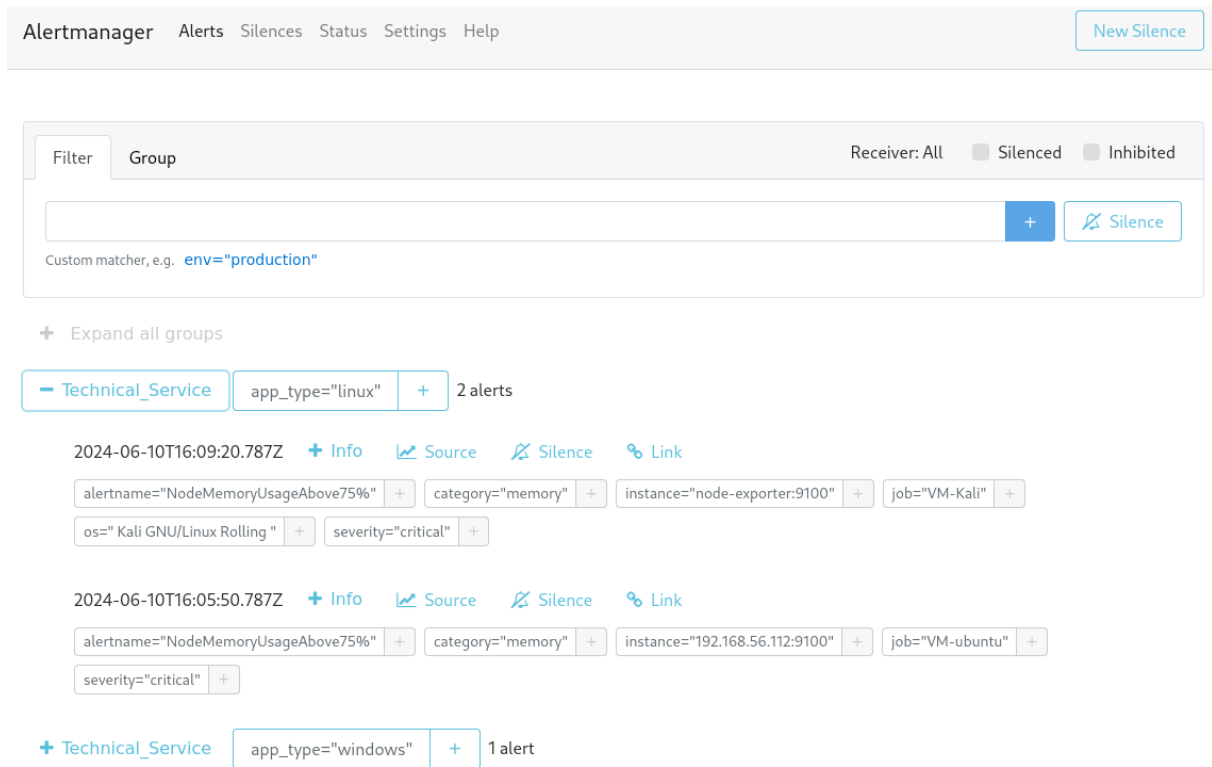


Figure 4.17: Fired Alerts Displayed in Alertmanager Interface

The figure illustrates that there are two firing alerts grouped by the `app_type` label 'linux,' which are `NodeMemoryUsageAbove60%` for both VM-Kali and VM-Ubuntu.

Additionally, the notifications of these fired grouped alerts have been sent to Gmail, in a single notification, as depicted in Figure. 4.18.

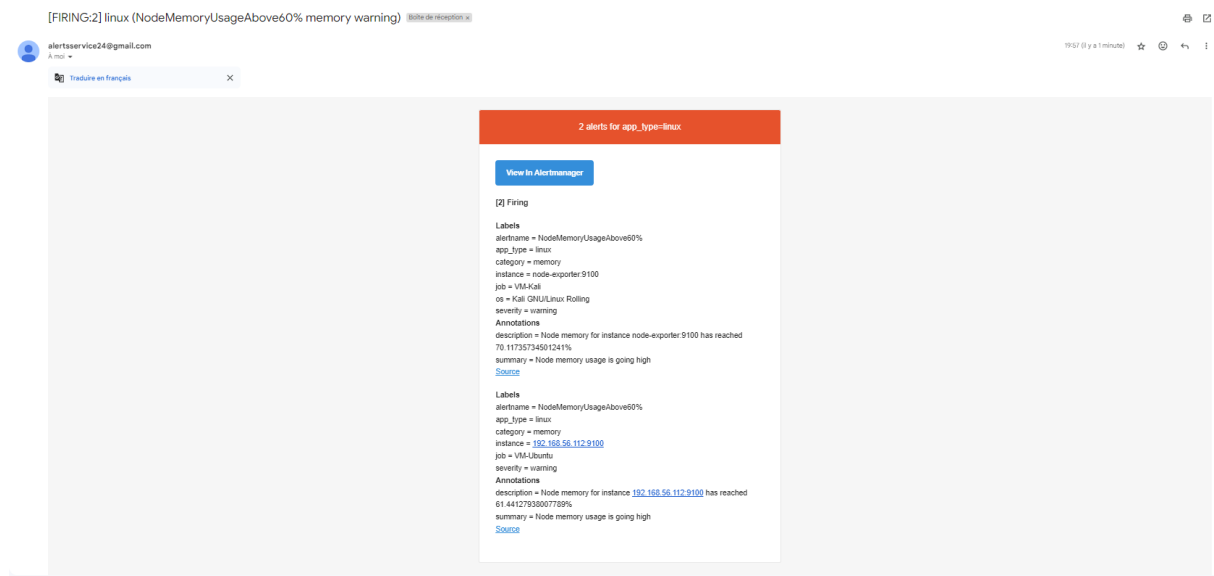


Figure 4.18: Gmail Notification for the Fired Alerts

4.3 Grafana Dashboards

In this section, we present the created dashboards. These dashboards are configured to be viewable by supervisors without editing privileges. From the Windows 10 VM, we log in using the supervisor account previously created in Grafana. The data visualizations for specific targets are displayed in the following dashboards.

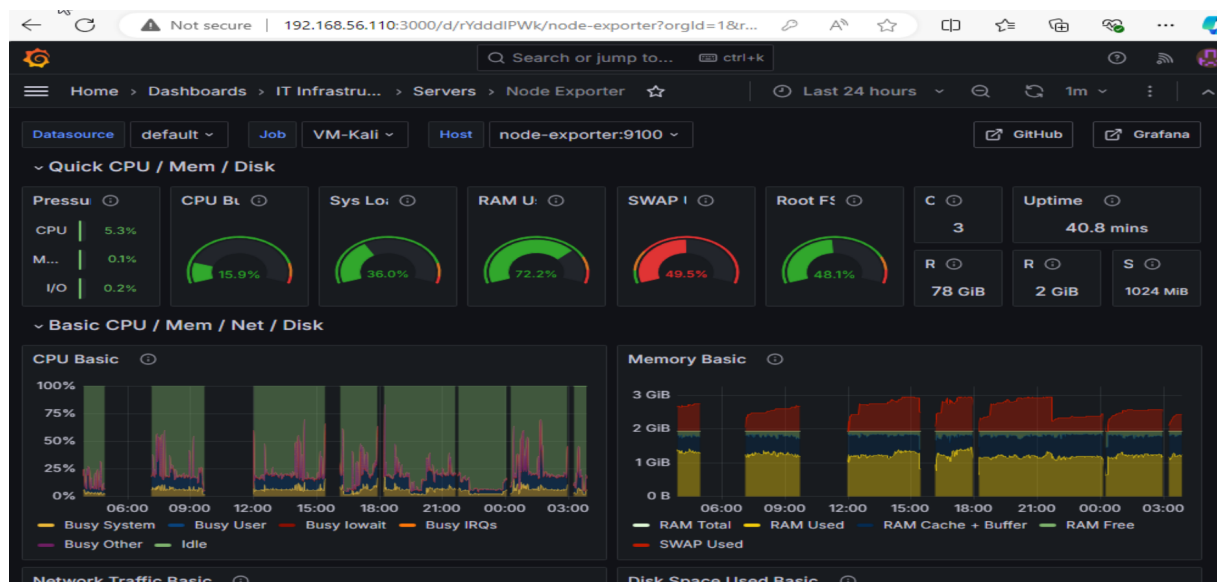


Figure 4.19: Data Visualisation for VM-Kali

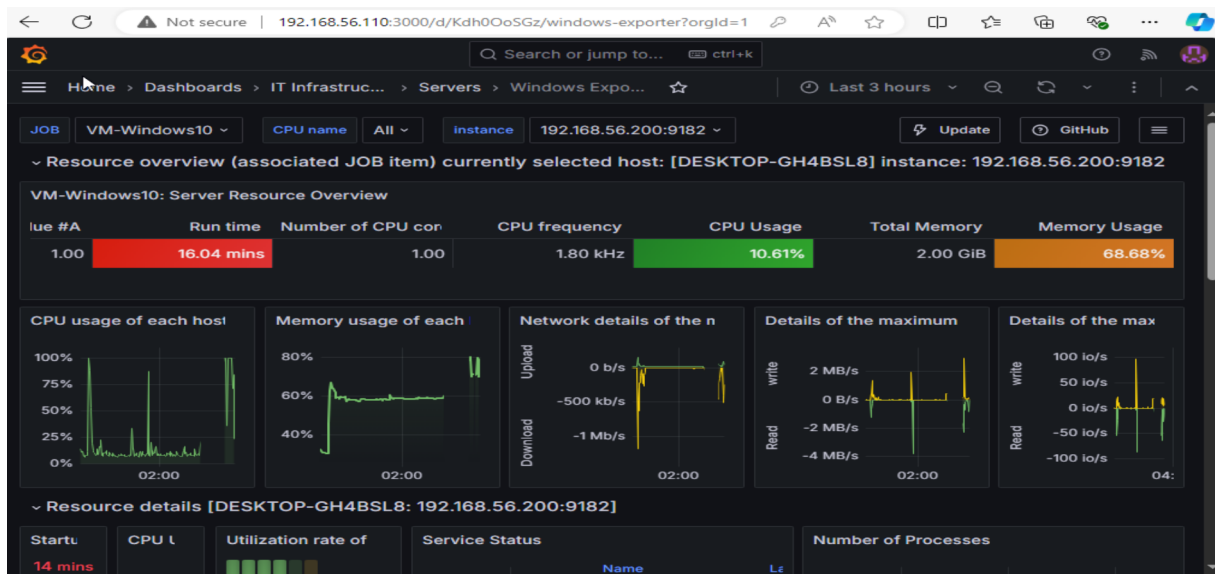


Figure 4.20: Windows VM Data Visualization Dashboard

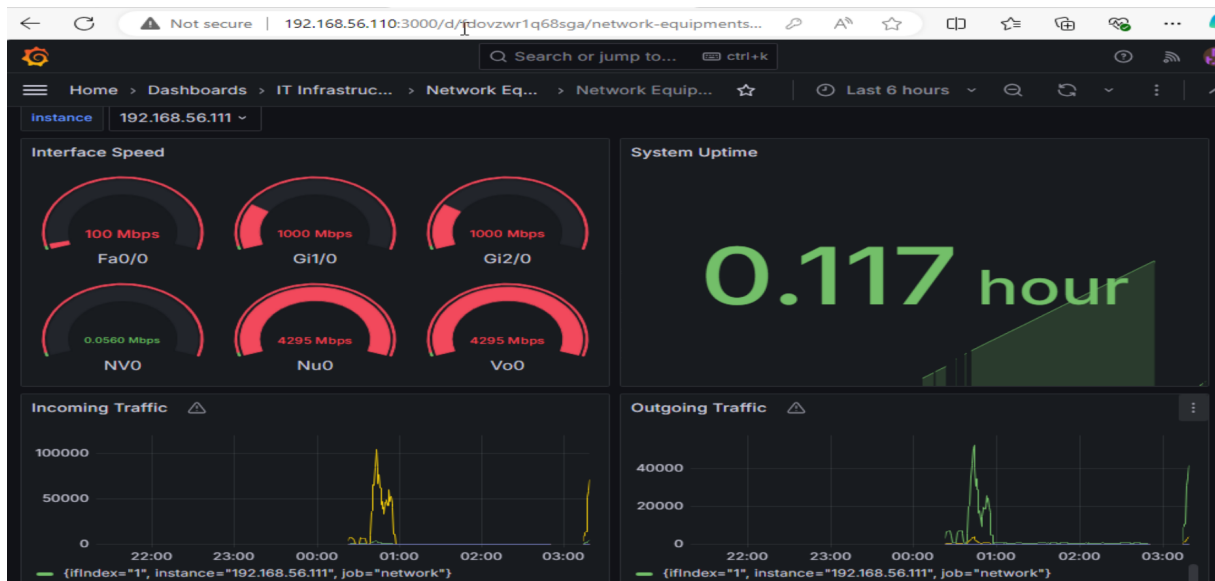


Figure 4.21: Data Visualisation for Router

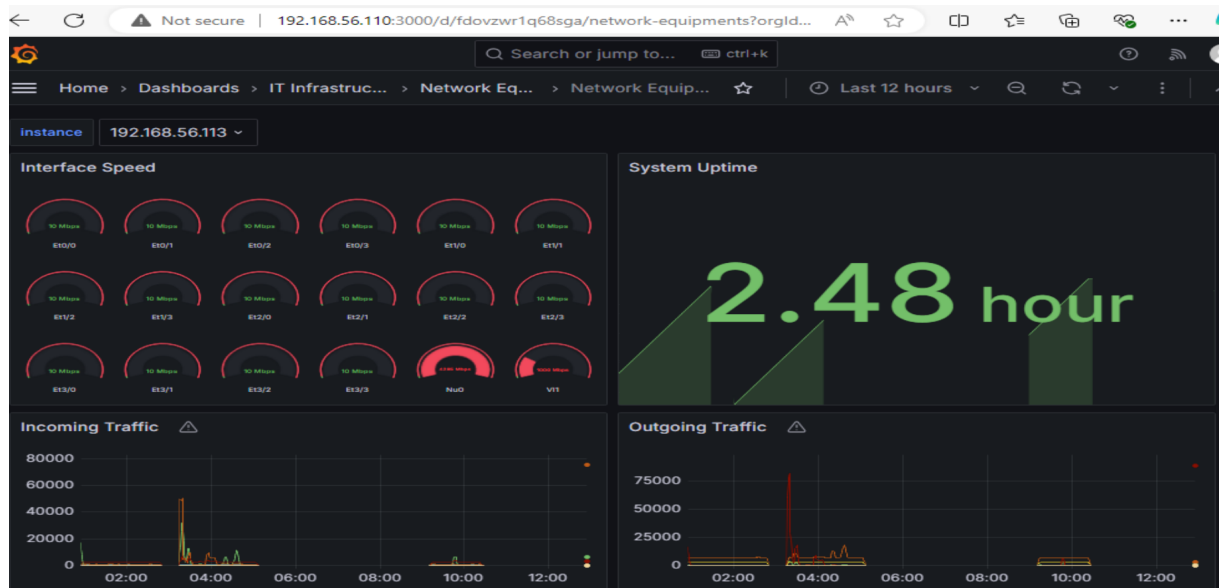


Figure 4.22: Data Visualisation for Switch

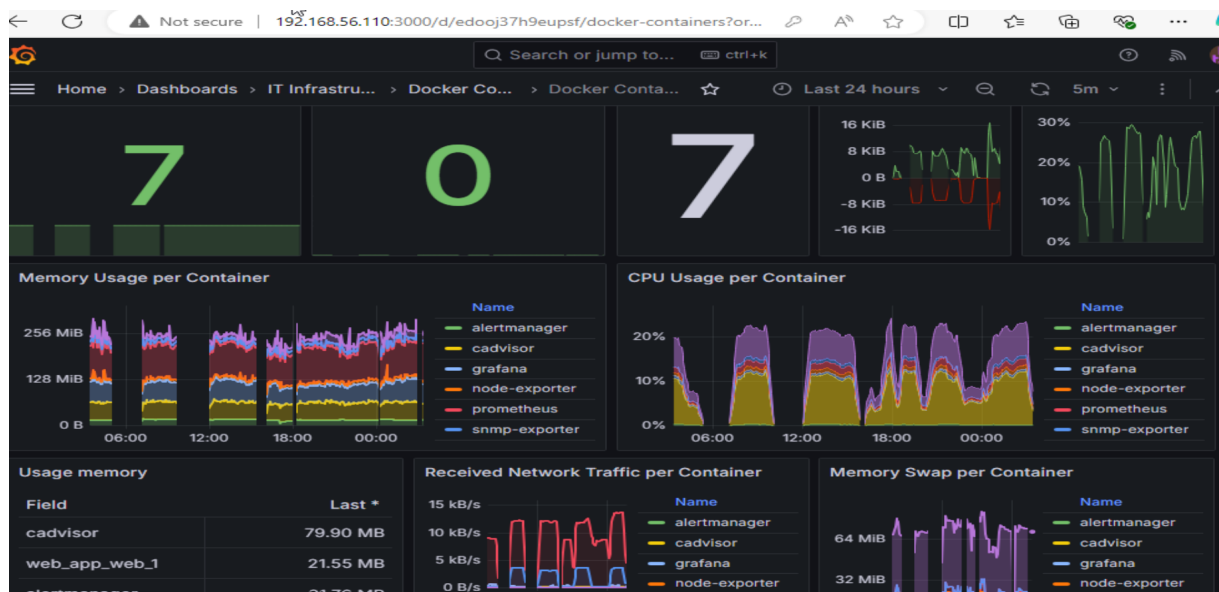


Figure 4.23: Data Visualisation for Docker

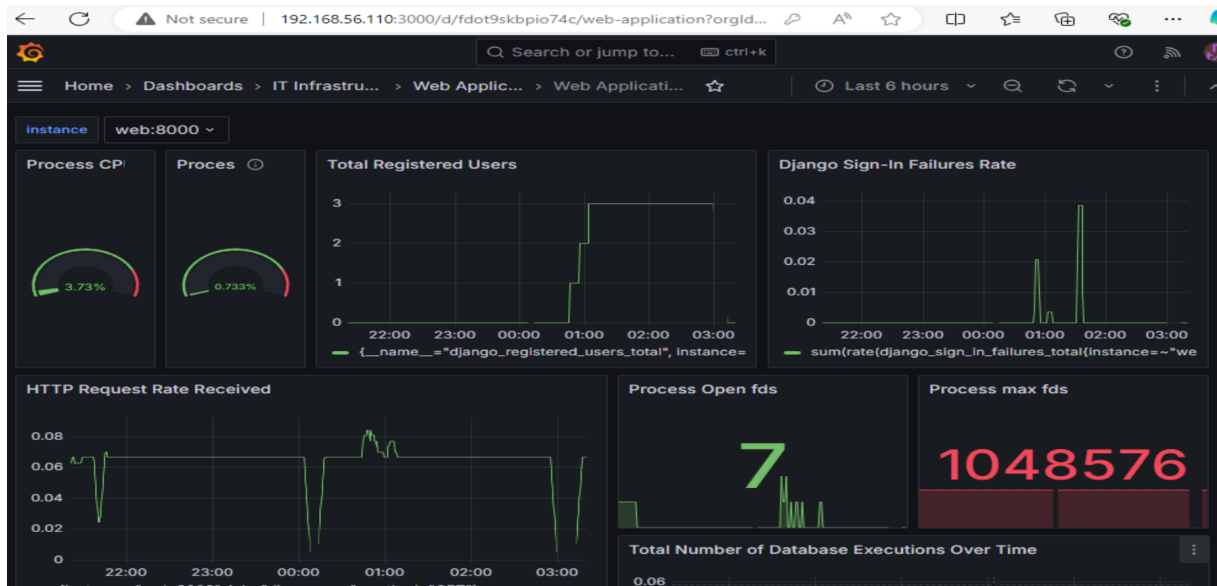


Figure 4.24: Data Visualisation for Web Application

After visualizing the result, we notice that the greatest advantage of using Prometheus and Grafana is their ability to monitor and visualize all aspects of IT infrastructure within a single, unified dashboard, regardless of the diverse technologies involved. Prometheus excels at collecting and storing metrics data from various sources, while Grafana provides powerful visualization tools to display this data in a coherent and customizable manner.

4.4 Evaluating of the Anomaly Detection Model

The final step in the anomaly detection using LSTM autoencoders involves evaluating the model's performance. This is done using several metrics, including Precision, Recall, F1 Score, and the Confusion Matrix. Precision measures the accuracy of the anomaly predictions, indicating how many of the detected anomalies are indeed actual anomalies. Recall evaluates the model's ability to find all actual anomalies within the dataset. The F1 Score provides a balanced measure of both Precision and Recall, which is particularly important in cases where the dataset may be imbalanced.

- **Precision**

Precision is the ratio of true positive predictions to the total predicted positives:

$$\text{Precision} = \frac{TP}{TP + FP}$$

where TP is the number of true positives and FP is the number of false positives. It indicates how many of the predicted anomalies are actual anomalies.

- **Recall**

Recall, also known as sensitivity, is the ratio of true positive predictions to the total actual positives:

$$\text{Recall} = \frac{TP}{TP + FN}$$

where TP is the number of true positives and FN is the number of false negatives. It measures the ability of the model to identify all actual anomalies.

- **F1 Score**

The F1 Score is the harmonic mean of precision and recall, providing a balance between the two metrics:

$$\text{F1 Score} = 2 \frac{\text{PrecisionRecall}}{\text{Precision} + \text{Recall}}$$

It is useful when the dataset is imbalanced, providing a single metric that balances precision and recall.

The model was tested on a testing dataset that contained two anomalies occurring at '2014-07-12 02:04:00' and '2014-07-14 21:44:00'. After setting the threshold for the reconstruction error, the model detected three anomalies, as depicted in Figure 4.25. We extracted the timestamps of these anomalies, shown in Figure 4.26, and visualized their occurrences within the testing dataset, illustrated in Figure 4.27.

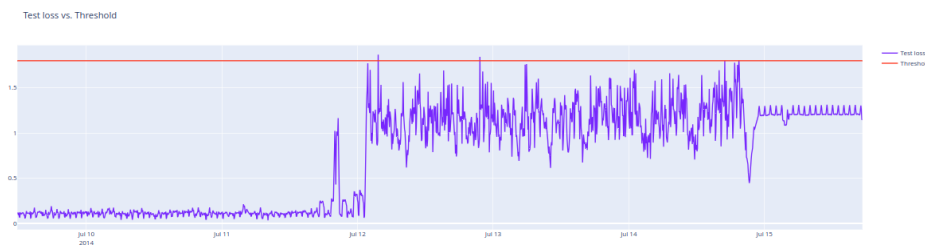


Figure 4.25: Detected Anomalies in Testing Dataset Using Reconstruction Error Threshold

```
[ ] anomalies
```

	loss	threshold	anomaly	value	time
17021	1.866606	1.798744	True	1.754357	2014-07-12 03:39:00
17237	1.841663	1.798744	True	1.754357	2014-07-12 21:39:00
17787	1.802144	1.798744	True	2.466349	2014-07-14 19:29:00

Figure 4.26: Timestamps of the detected Anomalies

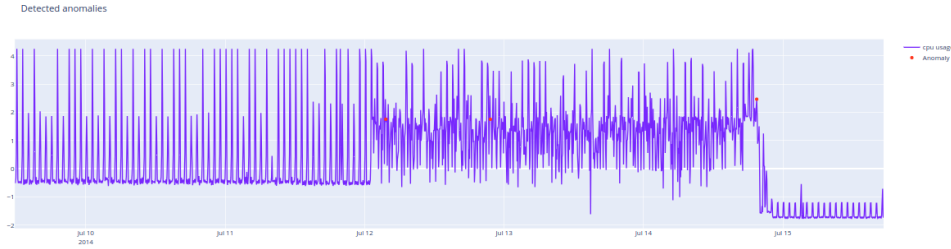


Figure 4.27: Occurrences of Detected Anomalies in the Testing Dataset

In Table 4.1, we provide the evaluation metrics for the model.

Parameter	Value
True Positives (TP)	2
False Positives (FP)	1
False Negatives (FN)	0
Precision	0.67
Recall	1.00
F1 Score	0.80

Table 4.1: Evaluation Metrics for Anomaly Detection

Here is the interpretation of the results:

The model correctly identified 2 true positives (TP), meaning it accurately flagged 2 instances as anomalies. However, it also produced 1 false positive (FP), where a non-anomalous instance was mistakenly classified as an anomaly. There were no false negatives (FN), indicating that the model did not miss any actual anomalies. The precision of the model is 0.67, which means that 67% of the instances identified as anomalies are indeed anomalies. The recall is 1.00, indicating that the model successfully identified all the anomalies present in the dataset. The F1 score, which is the harmonic mean of precision and recall, is 0.80. This score suggests that the model maintains a reasonably balanced performance between precision and recall.

The evaluation results indicate that the LSTM autoencoder model performs well in terms of recall, as it detected all the anomalies in the dataset. This is particularly important in scenarios where missing an anomaly could have significant consequences. However, the model does generate some false positives, as evidenced by the precision score of 0.67. This is likely due to the data imbalance, with only two anomalies in the entire dataset, making these anomalies rare.

4.5 Conclusion

In conclusion, the monitoring system setup using Prometheus and Grafana has proven to be highly effective in tracking and visualizing metrics across various technologies within the network architecture. The validation results confirm the successful configuration and operation of the system, highlighting its capability to provide real-time insights and ensure optimal performance. Furthermore, the evaluation of the LSTM autoencoder-based anomaly detection model demonstrates its ability in identifying anomalies in CPU usage. This robust monitoring and anomaly detection solution not only enhances visibility into the IT infrastructure but also facilitates proactive management and troubleshooting, thereby improving overall system reliability and efficiency.

4.6 Future Work

For our future work, enhancing the security of our monitoring system and integrating our LSTM autoencoder model for real-time anomaly detection are pivotal goals. These two approaches will significantly elevate the robustness and intelligence of our IT infrastructure monitoring.

Securing the Monitoring System: To ensure the security of our monitoring system, which currently leverages Prometheus and Grafana, we must implement comprehensive security measures. One effective approach is the deployment of a proxy server. A proxy server can act as an intermediary between the user and our monitoring system, filtering requests and blocking malicious traffic. This setup can help prevent unauthorized access and shield sensitive data from potential threats. Additionally, we should consider implementing HTTPS for encrypted communication, securing API endpoints with authentication and authorization mechanisms, and regularly updating our software to patch any vulnerabilities. Network segmentation and the use of firewalls can further isolate the monitoring system from other parts of the network, minimizing the risk of lateral movement in case of a breach. By integrating these security measures, we can protect our monitoring infrastructure against cyber threats and ensure the integrity and confidentiality of the collected data.

Integrating the Anomaly Detection Model in Real-Time: The integration of our LSTM autoencoder model for real-time anomaly detection with Prometheus and Grafana is the next critical step in enhancing our monitoring capabilities. This integration will enable us to automatically identify and respond to anomalies in time series data, thereby improving our proactive maintenance and incident response strategies. To achieve

this, we need to establish a data pipeline where Prometheus continuously collects metrics and forwards them to the anomaly detection model. This can be done by setting up a custom exporter or using a service like Kafka for real-time data streaming. Once the model processes the data and detects an anomaly, it can generate alerts that are fed back into Prometheus. Grafana can then visualize these alerts, providing real-time dashboards that highlight abnormal patterns and potential issues. This seamless integration will allow us to leverage the predictive capabilities of our model, enabling more intelligent and automated monitoring of our IT infrastructure.

By focusing on these two future work approaches, we can significantly enhance both the security and intelligence of our IT monitoring system, ensuring it is robust, secure, and capable of providing actionable insights in real time.

General Conclusion

In this thesis, we have presented the development and implementation of a comprehensive IT infrastructure monitoring system utilizing Prometheus and Grafana. The primary objective of the project was to create a robust and efficient monitoring solution capable of providing real-time insights into the health and performance of IT systems. By integrating advanced anomaly detection techniques using Long Short-Term Memory (LSTM) autoencoders, we aimed to enhance the system's ability to identify and respond to unusual patterns in time-series data, thereby improving overall reliability and operational efficiency.

Throughout the project, we encountered several challenges. The complexity of configuring Prometheus and integrating it with various exporters for specific metric collection required meticulous attention to detail. Additionally, the implementation of LSTM autoencoders for anomaly detection presented difficulties in tuning the model parameters and ensuring accurate detection without generating false positives. The deployment of the system using Docker Compose also required careful consideration to ensure scalability and consistent performance across different environments.

Despite these challenges, the project achieved significant results. The implemented monitoring system demonstrated its capability to provide real-time monitoring and alerting, with Prometheus effectively collecting and storing time-series data and Grafana offering powerful visualization tools. The integration of LSTM autoencoders for anomaly detection proved successful, allowing the system to identify subtle anomalies that traditional methods might overlook. The deployment strategy using Docker Compose ensured the system's scalability and ease of management, making it adaptable to various IT environments.

The success of this project opens up numerous opportunities for further development and enhancement. Future work could focus on improving the anomaly detection techniques, exploring more advanced machine learning models, enhancing the security of our monitoring system, and integrating our LSTM autoencoder model for real-time anomaly detection. These approaches will significantly elevate the robustness and intelligence of

our IT infrastructure monitoring.

In conclusion, this thesis has demonstrated the potential of combining Prometheus and Grafana with advanced anomaly detection techniques to create a powerful and scalable IT infrastructure monitoring system. While the project faced several challenges, the results achieved highlight the system's effectiveness and reliability. The ability for further development and enhancement ensures that this solution can continue to evolve, meeting the demands of increasingly complex IT environments and providing a robust framework for ensuring system health and performance.

Bibliography

- [1] Raghavendra Chalapathy and Sanjay Chawla. Deep learning for anomaly detection: A survey. *arXiv preprint arXiv:1901.03407*, 2019.
- [2] H Du Nguyen, Kim Phuc Tran, Sébastien Thomassey, and Moez Hamad. Forecasting and anomaly detection approaches using lstm and lstm autoencoder techniques with the applications in supply chain management. *International Journal of Information Management*, 57:102282, 2021.
- [3] Docker documentation. <https://docs.docker.com/config/daemon/prometheus>. Accessed 20 June 2024.
- [4] Ericsson blog,. <https://www.ericsson.com/en/blog/2023/11/how-to-automate-resource-dimensioning-in-cloud>. Accessed 20 June 2024.
- [5] “soundcloud, soundcloud blog. <https://developers.soundcloud.com/blog/prometheus-monitoring-at-soundcloud>. Accessed 20 June 2024.
- [6] Grafana, grafanalabs. <https://grafana.com/oss/prometheus>. Accessed 20 June 2024.
- [7] Octavian Mart, Catalin Negru, Florin Pop, and Aniello Castiglione. Observability in kubernetes cluster: Automatic anomalies detection using prometheus. In *2020 IEEE 22nd International Conference on High Performance Computing and Communications; IEEE 18th International Conference on Smart City; IEEE 6th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 565–570. IEEE, 2020.
- [8] Élisson da Silva Rocha, Leylane GF da Silva, Guto L Santos, Diego Bezerra, André Moreira, Glauco Gonçalves, Maria Valéria Marquezini, Amardeep Mehta, Mattias Wildeman, Judith Kelner, et al. Aggregating data center measurements for availability analysis. *Software: Practice and Experience*, 51(5):868–892, 2021.

- [9] Roland Mark Erdei and Laszlo Toka. Optimal resource provisioning for data-intensive microservices. In *NOMS 2022-2022 IEEE/IFIP Network Operations and Management Symposium*, pages 1–6. IEEE, 2022.
- [10] Nitin Sukhija and Elizabeth Bautista. Towards a framework for monitoring and analyzing high performance computing environments using kubernetes and prometheus. In *2019 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computing, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCCom/IOP/SCI)*, pages 257–262. IEEE, 2019.
- [11] Yue Zhao, Zain Nasrullah, and Zheng Li. Pyod: A python toolbox for scalable outlier detection. *Journal of machine learning research*, 20(96):1–7, 2019.
- [12] Pankaj Malhotra, Lovekesh Vig, Gautam Shroff, Puneet Agarwal, et al. Long short term memory networks for anomaly detection in time series. In *Esann*, volume 2015, page 89, 2015.
- [13] Atlassian. « it infrastructure: Definition components ». atlassian,. <https://www.atlassian.com/itsm/it-operations-management/it-infrastructure>. Accessed: 2024-05-19.
- [14] Janne Sirviö. Monitoring of a cloud-based it infrastructure. 2021.
- [15] James Turnbull. *The art of monitoring*. James Turnbull, 2014.
- [16] Marcello Cinque, Raffaele Della Corte, and Antonio Pecchia. Microservices monitoring with event logs and black box execution tracing. *IEEE transactions on services computing*, 15(1):294–307, 2019.
- [17] Radoslav Gatev. *Introducing Distributed Application Runtime (Dapr)*. Springer, 2021.
- [18] Julien Pivotto and Brian Brazil. *Prometheus: Up & Running*. " O'Reilly Media, Inc.", 2023.
- [19] Slawek Ligus. *Effective Monitoring and Alerting: For Web Operations*. " O'Reilly Media, Inc.", 2012.
- [20] Nagios xi | nagios. <https://www.nagios.com/products/nagios-xi/>. Accessed: 2024-06-16.

- [21] Jean-Philippe Martin-Flatin. Push vs. pull in web-based network management. In *Integrated Network Management VI. Distributed Management for the Networked Millennium. Proceedings of the Sixth IFIP/IEEE International Symposium on Integrated Network Management. (Cat. No. 99EX302)*, pages 3–18. IEEE, 1999.
- [22] Lei Chen, Ming Xian, and Jian Liu. Monitoring system of openstack cloud platform based on prometheus. In *2020 International Conference on Computer Vision, Image and Deep Learning (CVIDL)*, pages 206–209. IEEE, 2020.
- [23] Production-grade container orchestration. <https://kubernetes.io/>. Accessed: 2024-06-16.
- [24] Prometheus. overview | prometheus. <https://prometheus.io/docs/introduction/overview/>. Accessed: 2024-05-22.
- [25] “pagerduty | real-time operations | incident response | on-call.” pagerduty,. <https://www.pagerduty.com/>. Accessed: 2024-06-16.
- [26] Tiia Leppänen. Data visualization and monitoring with grafana and prometheus. 2021.
- [27] Prometheus. monitoring linux host metrics with the node exporter | prometheus. <https://prometheus.io/docs/guides/node-exporter/>. Accessed: 2024-05-21.
- [28] « promcat ». promcat,. <https://promcat.io/apps/wmi>. Accessed: 2024-05-21.
- [29] Prometheus. recording rules | prometheus. https://prometheus.io/docs/prometheus/latest/configuration/recording_rules/. Accessed: 2024-05-22.
- [30] Slack. « what is slack? » slack help center,. <https://slack.com/help/articles/115004071768-What-is-Slack->. Accessed: 2024-06-16.
- [31] Prometheus. alertmanager | prometheus. <https://prometheus.io/docs/alerting/latest/alertmanager/>. Accessed: 2024-05-21.
- [32] James Turnbull. *Monitoring with Prometheus*. Turnbull Press, 2018.
- [33] Prometheus. configuration | prometheus. <https://prometheus.io/docs/alerting/latest/configuration/>. Accessed: 2024-05-22.
- [34] Mike Julian. *Practical Monitoring: Effective Strategies for the Real World*. " O’Reilly Media, Inc.", 2017.

- [35] Mainak Chakraborty and Ajit Pratap Kundan. *Monitoring cloud-native applications: lead agile operations confidently using open source software*. Springer, 2021.
- [36] Simo Vuorinen. *Monitoring integration systems and visualization*. 2022.
- [37] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Outlier detection: A survey. *ACM Computing Surveys*, 14:15, 2007.
- [38] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly detection: A survey. *ACM computing surveys (CSUR)*, 41(3):1–58, 2009.
- [39] Mohiuddin Ahmed, Abdun Naser Mahmood, and Jiankun Hu. A survey of network anomaly detection techniques. *Journal of Network and Computer Applications*, 60:19–31, 2016.
- [40] Jason Brownlee. *Deep learning for time series forecasting: predict the future with MLPs, CNNs and LSTMs in Python*. Machine Learning Mastery, 2018.
- [41] Ane Blázquez-García, Angel Conde, Usue Mori, and Jose A Lozano. A review on outlier/anomaly detection in time series data. *ACM Computing Surveys (CSUR)*, 54(3):1–33, 2021.
- [42] Abdulmalik Shehu Yaro, Filip Maly, and Pavel Prazak. Outlier detection in time-series receive signal strength observation using z-score method with s n scale estimator for indoor localization. *Applied Sciences*, 13(6):3900, 2023.
- [43] Sepehr Maleki, Sasan Maleki, and Nicholas R Jennings. Unsupervised anomaly detection with lstm autoencoders using statistical data-filtering. *Applied Soft Computing*, 108:107443, 2021.
- [44] Chong Zhou and Randy C Paffenroth. Anomaly detection with robust deep autoencoders. In *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 665–674, 2017.
- [45] Zhaomin Chen, Chai Kiat Yeo, Bu Sung Lee, and Chiew Tong Lau. Autoencoder-based network anomaly detection. In *2018 Wireless telecommunications symposium (WTS)*, pages 1–5. IEEE, 2018.
- [46] Benjamin Lindemann, Benjamin Maschler, Nada Sahlab, and Michael Weyrich. A survey on anomaly detection for technical systems using lstm networks. *Computers in Industry*, 131:103498, 2021.

- [47] “machine learning service - amazon sagemaker - aws.” amazon web services, inc.
<https://aws.amazon.com/sagemaker/>. Accessed 20 June 2024.

Appendix A

Configuration Files

A.1 SNMP Configuration File

```
auths:
  router:
    version: 3
    security_level: authPriv
    auth_protocol: SHA
    username: snmpuser
    password: monitoring2024
    priv_protocol: AES
    priv_password: monitoring2024

  switch:
    version: 3
    security_level: authNoPriv
    auth_protocol: SHA
    username: snmpuser
    password: monitoring2024

modules:
  standard_mibs:
    walk:
      - 1.3.6.1.2.1.1 #sys_mibs
      - 1.3.6.1.2.1.2.2.1
```

```
- 1.3.6.1.2.1.31.1.1 # ifXTable
- 1.3.6.1.2.1.2.2.1.10 # ifInOctets
- 1.3.6.1.2.1.2.2.1.16 # ifOutOctets
metrics:

- name: ifInOctets
  oid: 1.3.6.1.2.1.2.2.1.10
  type: counter
  help: "Incoming octets on the interface"
  indexes:
    - labelname: ifIndex
      type: gauge
    - labelname: ifDescr
      type: DisplayString

- name: ifOutOctets
  oid: 1.3.6.1.2.1.2.2.1.16
  type: counter
  help: "Outgoing octets on the interface"
  indexes:
    - labelname: ifIndex
      type: gauge
    - labelname: ifDescr
      type: DisplayString

- name: ifOperStatus
  oid: 1.3.6.1.2.1.2.2.1.8
  type: gauge
  help: "Interface operational status"
  indexes:
    - labelname: ifIndex
      type: gauge

- name: ifIndex
  oid: 1.3.6.1.2.1.2.2.1.1
  type: gauge
```

```
indexes:
- labelname: ifIndex
  type: Integer
lookups:
- labels:
  - ifIndex
  labelname: ifDescr
  oid: 1.3.6.1.2.1.2.2.1.2
  type: DisplayString
- labels:
  - ifIndex
  labelname: ifName
  oid: 1.3.6.1.2.1.31.1.1.1.1
  type: DisplayString
- labels:
  - ifIndex
  labelname: ifAlias
  oid: 1.3.6.1.2.1.31.1.1.1.18
  type: DisplayString

- name: ifSpeed
  oid: 1.3.6.1.2.1.2.2.1.5
  type: gauge
  indexes:
  - labelname: ifIndex
    type: Integer
  lookups:
  - labels:
    - ifIndex
    labelname: ifDescr
    oid: 1.3.6.1.2.1.2.2.1.2
    type: DisplayString
  - labels:
    - ifIndex
    labelname: ifName
    oid: 1.3.6.1.2.1.31.1.1.1.1
```

```
    type: DisplayString

- name: sysUpTime
  oid: 1.3.6.1.2.1.1.3
  type: counter
  lookups:
  - labels:
    labelname: sysDescr
    oid: 1.3.6.1.2.1.1.1.0
    type: DisplayString
  - labels:
    labelname: sysName
    oid: 1.3.6.1.2.1.1.5.0
    type: DisplayString
  - labels:
    labelname: sysLocation
    oid: 1.3.6.1.2.1.1.6.0
    type: DisplayString
  - labels:
    labelname: sysContact
    oid: 1.3.6.1.2.1.1.4.0
    type: DisplayString
```

A.2 Docker Compose Configuration File

```
version: '3.7'

services:
  prometheus:
    image: prom/prometheus:latest
    container_name: prometheus
    volumes:
      - ./prometheus:/etc/prometheus
```

```
- prom_data:/prometheus
command:
- '--config.file=/etc/prometheus/prometheus.yml'
- '--storage.tsdb.path=/prometheus'
- '--web.enable-lifecycle'
ports:
- 9090:9090
restart: always
networks:
- monitoring

grafana:
image: grafana/grafana:latest
container_name: grafana
volumes:
- ./grafana:/etc/grafana/provisioning/datasources
- grafana-data:/var/lib/grafana
ports:
- 3000:3000
environment:
- GF_SECURITY_ADMIN_PASSWORD=admin
- GF_SECURITY_ADMIN_USER=admin
- GF_METRICS_ENABLED=true
restart: always
networks:
- monitoring

alertmanager:
image: prom/alertmanager:latest
container_name: alertmanager
volumes:
- ./alertmanager:/etc/alertmanager
command:
- '--config.file=/etc/alertmanager/alertmanager.yml'
ports:
- 9093:9093
```



```
restart: always
networks:
  - monitoring

snmp-exporter:
  image: prom/snmp-exporter:latest
  container_name: snmp-exporter
  volumes:
    - ./snmp-exporter/snmp.yml:/etc/snmp-exporter/snmp.yml
  command:
    - '--config.file=/etc/snmp-exporter/snmp.yml'
  ports:
    - 9116:9116
  restart: always
  networks:
    - monitoring

node-exporter:
  image: prom/node-exporter:latest
  container_name: node-exporter
  ports:
    - 9100:9100
  restart: always
  volumes:
    - /proc:/host/proc:ro
    - /sys:/host/sys:ro
    - /:/rootfs:ro
  command:
    - '--path.procfs=/host/proc'
    - '--path.rootfs=/rootfs'
    - '--path.sysfs=/host/sys'
  networks:
    - monitoring

cadvisor:
  image: gcr.io/cadvisor/cadvisor:latest
```

```
container_name: cadvisor
ports:
  - "8080:8080"

volumes:
  - /:/rootfs:ro
  - /var/run:/var/run:ro
  - /sys:/sys:ro
  - /var/lib/docker/:/var/lib/docker:ro
  - /dev/disk/:/dev/disk:ro
devices:
  - /dev/kmsg:/dev/kmsg

restart: always
privileged: true
networks:
  - monitoring

networks:
  monitoring:
    external: true

volumes:
  prom_data:
  grafana-data:
```

A.3 Alerting Rules Configuration File

- Linux Rules

```
groups:
  - name: linux-rules
    rules:
```

```
- alert: NodeExporterDown
  expr: (up{job="VM-Kali"} == 0 or up{job="VM-Ubunto"} == 0)
  for: 2m
  labels:
    severity: critical
    app_type: linux
    category: server
  annotations:
    summary: "Node Exporter is down"
    description: "Node Exporter is down for more than 2 minutes"

- record: job:node_memory_Mem_bytes:available
  expr: (node_memory_MemAvailable_bytes /
↪ node_memory_MemTotal_bytes) * 100

- alert: NodeMemoryUsageAbove60%
  expr: 60 < (100 - job:node_memory_Mem_bytes:available) < 75
  for: 2m
  labels:
    severity: warning
    app_type: linux
    category: memory
  annotations:
    summary: "Node memory usage is going high"
    description: "Node memory for instance {{ $labels.instance
↪ }} has reached {{ $value }}%"

- alert: NodeMemoryUsageAbove75%
  expr: (100 - job:node_memory_Mem_bytes:available) >= 75
  for: 2m
  labels:
    severity: critical
    app_type: linux
    category: memory
  annotations:
```

```
    summary: "Node memory usage is very HIGH"
    description: "Node memory for instance {{ $labels.instance
↪ }} has reached {{ $value }}%"

- alert: NodeCPUUsageHigh
  expr: 100 - (avg by(instance)
↪ (irate(node_cpu_seconds_total{mode="idle"}[1m])) * 100) > 80
  for: 2m
  labels:
    severity: critical
    app_type: linux
    category: cpu
  annotations:
    summary: "Node CPU usage is HIGH"
    description: "CPU load for instance {{ $labels.instance }}
↪ has reached {{ $value }}%"
```

- Windows Rules

```
groups:
- name: windows-rules
  rules:

- alert: WMIExporterDown
  expr: up{job="VM-Windows10"} == 0
  for: 2m
  labels:
    severity: critical
    app_type: windows
    category: target
  annotations:
    summary: "WMI Exporter is down"
    description: "WMI Exporter is down for more than 2 minutes"

- record: job:wmi_physical_memory_bytes:free
```

```
    expr: (wmi_os_physical_memory_free_bytes /
↪ wmi_cs_physical_memory_bytes) * 100

- alert: WindowsMemoryUsageAbove60%
  expr: 60 < (100 - job:wmi_physical_memory_bytes:free) < 75
  for: 2m
  labels:
    severity: warning
    app_type: windows
    category: memory
  annotations:
    summary: "Windows memory usage is going high"
    description: "Windows memory for instance {{
↪ $labels.instance }} has left only {{ $value }}%"

- alert: WindowsMemoryUsageAbove75%
  expr: (100 - job:wmi_physical_memory_bytes:free) >= 75
  for: 2m
  labels:
    severity: critical
    app_type: windows
    category: memory
  annotations:
    summary: "Windows memory usage is HIGH"
    description: "Windows memory for instance {{
↪ $labels.instance }} has left only {{ $value }}%"

- alert: WindowsCPUUsageHigh
  expr: 100 - (avg by (instance)
↪ (rate(wmi_cpu_time_total{mode="idle"}[1m])) * 100) > 80
  for: 2m
  labels:
    severity: warning
    app_type: windows
    category: cpu
  annotations:
```

```
summary: "Windows CPU usage is HIGH"
description: "CPU load for instance {{ $labels.instance }}
↪ has reached {{ $value }}"
```

- **Web Application Rules**

```
groups:
- name: app-rules
  rules:
- alert: SignInFailuresHigh
  expr: sum(rate(django_sign_in_failures_total[5m])) > (10/300)
↪ # 10 failures / 5 min = 300 sec
  for: 2m
  labels:
    severity: warning
    app_type: django
  annotations:
    summary: "High number of failed sign-in attempts"
    description: "The rate of failed sign-in attempts in Django"
↪ has exceeded the threshold for instance {{ $labels.instance }}"
    app_link: 'http://127.0.0.1:8000/'
```

A.4 Alertmanager Configuration File

```
global:
  smtp_from: 'alertsservice24@gmail.com'
  smtp_smarthost: smtp.gmail.com:587
  smtp_auth_username: 'alertsservice24@gmail.com'
```

```
smtp_auth_identity: 'alertsservice24@gmail.com'
smtp_auth_password: 'vykc auph bqmi yaru'

route:
  group_by: ['app_type']
  group_wait: 10s
  group_interval: 5m
  repeat_interval: 3h
  receiver: Fallback_receiver

routes:
  - match:
      app_type: linux
      receiver: Technical_Service
      group_wait: 30s
      group_interval: 5m
      repeat_interval: 1h

  - match:
      app_type: windows
      receiver: Technical_Service
      group_wait: 30s
      group_interval: 5m
      repeat_interval: 1h

  - match:
      app_type: django
      receiver: Application_Service
      group_wait: 30s
      group_interval: 5m
      repeat_interval: 1h

receivers:
  - name: Fallback_receiver
    email_configs:
      - to: 'fallbackreceiver@gmail.com'
```

```
- name: Technical_Service
  email_configs:
    - to: 'technicalservice@gmail.com'

- name: Application_Service
  email_configs:
    - to: 'applicationservice@gmail.com'

inhibit_rules:
  - source_match:
      severity: 'critical'
    target_match:
      severity: 'warning'
    equal: ['app_type', 'category']

\end{itemize}
```