# POLITECNICO DI TORINO

## Master's Degree in Data Science & Engineering

### Master's Degree Thesis

# Centralized Monitoring Infrastructure on Cloud: An Open Source Approach.

**Supervisors**

**Prof. Daniele APILETTI**

**Eng. Davide BURDESE**

**Eng. Alessandro TEDESCO**

**Candidate**

**Niccolò DIMONTE**

**April 2024**

# Summary

Navigating the contemporary digital landscape requires adept management of vast data volumes, prompting the need for robust architectures and innovative solutions. This challenge is particularly evident in establishing a comprehensive monitoring system for seamless operations and service optimization. Systematic monitoring processes are indispensable for timely issue detection, minimizing downtime, and bolstering overall system reliability.

The broader challenge encompasses tracking and analyzing resource utilization for effective cost management strategies. Crucial metrics, such as resource consumption, play a pivotal role in judiciously using client investments, optimizing both operational efficiency and fiscal responsibility. Comprehensive monitoring protocols contribute to efficient resource management by anticipating and addressing capacity requirements, facilitating informed decisions in resource provisioning, and fostering a scalable and adaptive infrastructure.

Within this context, collaboration with Mediamente Consulting, a consultancy firm specializing in Oracle technologies, has been invaluable. Their commitment to monitoring extends beyond issue detection, cost management, and resource optimization, forming the cornerstone of dedication to delivering reliable, high-quality services. The firm leverages cutting-edge monitoring technologies to proactively address challenges, ensuring unparalleled support for clients. Insights gleaned from monitoring practices serve as a foundation for issue resolution and empower the consultancy team to make informed

recommendations, driving continuous improvement and innovation. This thesis focuses on revolutionizing incident monitoring systems within a firm specializing in Oracle database management. The existing challenge lies in the cumbersome process of connecting via VPN to individual customer networks when incidents are reported, especially given the multitude of clients and targets. The proposed solution involves a Proof of Concept (PoC) centered around a centralized architecture with Prometheus Agents installed on customer targets. These agents utilize remote-write to send real-time data to a centralized time-series database.

The detailed thesis activities involve several key steps, starting with carefully designing the architecture. This includes not just outlining the structure but also the detailed process of setting up Prometheus. Another important aspect is developing a flexible Python exporter for custom queries, adding adaptability to the overall solution.

Furthermore, the implementation phase incorporates the setup of Grafana Mimir, where special attention is given to configuring remote write capabilities. This integration aims to facilitate seamless data transfer, enabling efficient collaboration and real-time incident monitoring across distributed components. In the pursuit of a scalable and efficient solution, the thesis underscores the development of customized Grafana dashboards tailored for each customer. Noteworthy is the optimization strategy employed, wherein these dashboards are designed to load only the essential data, ensuring a streamlined and client-specific incident monitoring experience. This meticulous approach seeks to provide not just a functional but a highly tailored and user-friendly solution for effective data visualization and incident management. In summary, the evolving landscape of technology and data management necessitates the development of alternative open-source solutions to efficiently handle complexities in monitoring resources and detecting potential problems. The collaboration with Mediamente Consulting underlines the commitment to provide optimal

value and satisfaction to customers in relation to these challenges.

# Acknowledgements

ACKNOWLEDGMENTS

*Volevo ringraziare la mia famiglia, i miei amici, i miei colleghi e me stesso.*

# Table of Contents

**List of Tables** X

**List of Figures** XI

**Acronyms** XIV

**1 Introduction** 1

1.1 Company . . . 2

1.2 Case Study . . . 3

1.3 Project Structure and Workflow . . . 4

**2 Background** 7

2.1 Time Series and Database Timeseries . . . 7

2.2 Oracle Database . . . 9

2.2.1 Database and Instance . . . 9

2.2.2 Data Storage and Architecture . . . 10

2.3 Oracle Enterprise Manager (EM) . . . 12

2.3.1 Monitoring Utilities . . . 12

2.3.2 Oracle Enterprise Manager Tables for Metrics . . . 13

2.4 Prometheus . . . 13

2.4.1 Prometheus remote write . . . 15

2.5 Grafana . . . 17

2.6 Mimir . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18

**3 Methodology** 21

3.1 Architecture . . . . . . . . . . . . . . . . . . . . . . . . . . 21

3.1.1 General Architecture . . . . . . . . . . . . . . . . . . 21

3.1.2 Subnet structure . . . . . . . . . . . . . . . . . . . . 22

3.1.3 Role of the Container . . . . . . . . . . . . . . . . . . 23

3.1.4 Architecture Objectives . . . . . . . . . . . . . . . . . 23

3.1.5 Role of Prometheus and Mimir in the Architecture . . 23

3.1.6 Using Grafana for Data Visualisation . . . . . . . . . . 24

3.2 Configuration . . . . . . . . . . . . . . . . . . . . . . . . . . 24

3.2.1 Problem of Access to the Oracle Database and Enterprise Manager . . . . . . . . . . . . . . . . . . . . . . . . . 24

3.2.2 Solution: Python Exporter for Oracle DB . . . . . . . 25

3.2.3 Custom Python Exporter for Oracle DB . . . . . . . . 25

3.2.4 The configuration file . . . . . . . . . . . . . . . . . . 27

**4 Proposed Solution** 30

4.1 How Prometheus works and how Node Exporter interact with 30

4.1.1 How Prometheus Scraping Works . . . . . . . . . . . . 31

4.1.2 Target Selection . . . . . . . . . . . . . . . . . . . . . 31

4.1.3 Scraping process for Node Exporter . . . . . . . . . . . 32

4.2 Remote Write and Mimir . . . . . . . . . . . . . . . . . . . . 33

4.2.1 Mimir object storage and ring configuration . . . . . . 34

4.3 Data Manipulation and Visualization with Grafana . . . . . . 35

4.3.1 Data Source . . . . . . . . . . . . . . . . . . . . . . . . 36

4.3.2 PromQL and metrics . . . . . . . . . . . . . . . . . . . 36

4.3.3 Dashboard and global variables . . . . . . . . . . . . . 37

**5 Visualization and Results** 41

5.1 Visualization . . . . . . . . . . . . . . . . . . . . . . . . . . 41

5.2 Results . . . . . . . . . . 44

**6 Conclusions** 48

6.1 Future Works . . . . . . . . . . 49

**A Node Exporter** 50

**Bibliography** 57

# List of Tables

4.1 Labels extracted with SELECT in metrics' query . . . . . . . . 32

# List of Figures


2.1 Oracle Database Architecture [1] . . . . . . . . . . . . . . . . . 10
2.2 Oracle Database Instance Architecture [2] . . . . . . . . . . . . 11
2.3 Prometheus Time Series structure [5] . . . . . . . . . . . . . . . 15

3.1 The architecture of proposed solution. . . . . . . . . . . . . . . . 22
3.2 A *config.yml* snapshot . . . . . . . . . . . . . . . . . . . . . . 28

4.1 Example of different timeseries . . . . . . . . . . . . . . . . . . . 31
4.2 An example of metrics configuration for monitoring . . . . . . 33
4.3 Data-source on Grafana . . . . . . . . . . . . . . . . . . . . . . . 36
4.4 Example of cpu% timeseries . . . . . . . . . . . . . . . . . . . . . 37
4.5 Global Variables for 'Host' dashboard. . . . . . . . . . . . . . . 38
4.6 Example of global variables. . . . . . . . . . . . . . . . . . . . . . 39
4.7 Variables selection on dashboard. . . . . . . . . . . . . . . . . . . 39

5.1 Timeseries and stats visualization example. . . . . . . . . . . . 42
5.2 Different colors can be used to alert a critical threshold. . . . . 42
5.3 Table visualization example. . . . . . . . . . . . . . . . . . . . . . 43
5.4 Filesystems monitoring panels. . . . . . . . . . . . . . . . . . . . 44
5.5 Panels for health's device monitoring. . . . . . . . . . . . . . . . 44
5.6 Disk Group Usage info. . . . . . . . . . . . . . . . . . . . . . . . 45
5.7 Tablespaces info. . . . . . . . . . . . . . . . . . . . . . . . . . . . 45

5.8 Datafiles path info. . . . . . . . . . . . . . . . . . . . . . . . . . . 46

# Acronyms

**DB**

Database

**EM**

Enterprise Manager

**IT**

Information Technology

**PoC**

Proof of Concept

# Chapter 1

# Introduction

In the digital age in which we live, Information Technology (IT) plays a fundamental role in every aspect of our daily lives, both personal and professional. IT environments have become complex and interconnected, with networks, systems and applications supporting a wide range of activities.
In this context, systems monitoring emerges as an essential practice to ensure the proper functioning, security and efficiency of IT infrastructures. Systems monitoring, or simply 'monitoring', refers to the process of continuously supervising and evaluating the performance, status and behaviour of the components of an IT system. These components may include servers, networks, databases, applications and other hardware and software devices critical to business operations. In this thesis work, we will mainly discuss database monitoring.
Through the use of specialised tools, monitoring collects real-time or near real-time data, analysing key metrics and reporting any anomalies or problems. The importance of systems monitoring in IT is evident on several fronts. First, monitoring provides complete visibility into the IT environment, enabling system administrators and network operators to quickly identify and resolve any malfunctions or problems before they can cause disruptions or data loss.

In addition, monitoring helps optimise IT resources, identifying inefficiencies and opportunities for performance improvement. In summary, systems monitoring is a fundamental component of modern IT management, offering crucial benefits in terms of performance, security and reliability. This thesis aims to explore in detail the importance of systems monitoring, examining its various applications, best practices and related challenges, in order to provide a comprehensive view of this essential aspect of contemporary IT management.

## 1.1 Company

The focal point of this thesis revolves around the endeavors of Mediamente Consulting S.r.l., a prominent entity within the IT consultancy sector renowned for its specialization in advanced business analytics. Originating as an innovative startup within the i3P incubator of Politecnico di Torino back in 2013, the company swiftly gained traction, culminating in its exit from the incubator in 2016. Today, it stands as an integral component of Var Group (SeSa S.p.A. group), seamlessly integrated into its Data Science business unit.

Mediamente Consulting prides itself on a diverse portfolio structured around five distinct business units: Corporate Performance Management, Advanced Analytics, Business Intelligence, Data Integration and Management, and Technological Infrastructure. Particularly noteworthy is the pivotal role played by the Technological Infrastructure unit, serving as the linchpin of the company's operations.

This unit spearheads a wide array of activities, ranging from consultancy services tailored to various Oracle technologies, including engineered systems, to the evaluation and optimization of architectural performance, to the ongoing management and modernization of technological infrastructure, whether on-premise or in the cloud.

The research for this thesis is closely connected to the work done in the Technological Infrastructure business unit. This close relationship highlights a joint effort to enhance the company's skills and capabilities in the important field of IT consultancy.

## 1.2 Mediamente Consulting Case Study

The systems monitoring performed by Mediamente Consulting, although a vital component of its services, faces a number of significant challenges that undermine the efficiency and effectiveness of the entire process. These include three main issues that require careful consideration and resolution: the obsolescence of the system in use, the complexity of the client's network access process, and security concerns.

Firstly, the current monitoring system employed by Mediamente Consulting is based on PHP scripts that update detected incidents via events. However, this solution is plagued by obsolescence, as the scripts may not be up to modern system monitoring requirements. Lack of updates and reliance on outdated technologies could compromise the effectiveness of monitoring and limit Mediamente Consulting's ability to detect and resolve client issues in a timely manner.

Secondly, the process of accessing the client's network is a maze of complexity and cumbersomeness. To carry out monitoring, Mediamente Consulting's consultants have to overcome several hurdles, including obtaining remote shell keys, access via VPN and connection to the client's Enterprise Manager. These procedures require considerable time and resources, slowing down Mediamente Consulting's ability to provide quick and efficient support to its clients.

Finally, there are security concerns that gravitate around the client's network access process. Each client has its own security keys, but there is no guarantee that all consultants in the company have access to all the necessary keys. This disparity can lead to delays and interruptions in the monitoring process, as consultants may find themselves in the position of having to request missing keys every time there is a client alert.

In summary, the problems in Mediamente Consulting's current monitoring system pose a significant challenge to the overall effectiveness of its services. Tackling these problems requires a holistic and innovative approach aimed at modernising the existing infrastructure, simplifying access processes and ensuring high security in customer interaction.

## 1.3 Project Structure and Workflow

The project structure was outlined through several consecutive phases, each aimed at achieving a specific goal in the development process of the multi-target monitoring system. Initially, we conducted an in-depth study of the current state of monitoring technologies and available open source tools, as well as the architecture of the Oracle Enterprise Manager and the data structure for queries.

Subsequently, we divided the project into several parts, each of which explored a key aspect of the development process. In the following chapters, we addressed the following areas:

1. **Background**: This chapter provided an in-depth overview of the concepts and technologies relevant to understanding the solutions adopted in the project. This included fundamental monitoring concepts, the analysis of customer requirements and an overview of the open source technologies

available for system monitoring.

2. **Network infrastructure development**: In this phase, we focused on the implementation of the network infrastructure required for the operation of the monitoring system. This included the configuration of VPNs for remote access to customer systems and the creation of a secure network for the transfer of monitoring data.

3. **Configuration and data retrieval from the Oracle Enterprise Manager (EM)**: We devoted a chapter to analysing the configuration and retrieval of data via queries from the Oracle Enterprise Manager. This included the definition of the queries needed to extract the relevant information from the monitored systems.

4. **Manipulation of data into time series**: This phase involved the manipulation of data obtained from Oracle Enterprise Manager queries into time series that could be used for analysis and visualisation. We explored techniques to transform and process the raw data into formats suitable for analysis.

5. **Dashboard creation**: Finally, we developed customised dashboards for visualising monitoring data. This included the design and implementation of charts and tables to clearly represent the metrics of interest to customers.

Through these steps, we were able to develop a comprehensive and scalable monitoring system that meets our customers' needs, providing them with an effective solution for monitoring their systems.

# Chapter 2

# Background

## 2.1 Time Series and Database Timeseries

**Time Series and Time Series-Based Data**

Time series are a type of data that record observations sequentially over time. These observations may be collected at regular or irregular intervals, and represent variations or measurements of phenomena over time. For example, they may include data such as the temperature recorded every hour, share prices recorded every day, the number of visitors to a website every minute, and so on.

These data are widely used in a wide range of industries and applications. Examples include:

1. **Prediction and Analysis:** Time series can be used to predict the future behaviour of a phenomenon, such as share price, traffic on a network, or demand for a product.

2. **Monitoring and Control:** They can be used to continuously monitor systems and identify any anomalies or changes in data behaviour, such as in monitoring the performance of a computer system or analysing sensor data.

3. **Anomaly Detection:** Can be used to identify anomalous patterns in the data, signalling situations outside the norm that require human attention or intervention.

4. **Process Optimisation:** Time series can be analysed to identify trends and patterns that can be used to optimise processes and improve performance.

**Time Series Databases**

Time series databases are specially designed to manage and store large volumes of time series-based data efficiently. These databases are optimised to support operations such as inserting, updating, retrieving and analysing time series data.

Typically, time series databases use a timestamp-value based data model (key-value timestamp-value). In this model, each data point is associated with a timestamp indicating when it was recorded and a value representing the observation or measurement made at that time. This data is organised and indexed in a way that allows quick and efficient access based on the timestamp.

The use of time series databases offers several advantages:

1. **Archive Efficiency:** Time series databases are optimised to store large volumes of time data efficiently, reducing data storage and management costs.

2. **Query Speed:** Thanks to their optimised structure, time series databases enable fast queries and real-time analysis of temporal data, supporting immediate and responsive decisions.

3. **Scalability:** These databases are designed to handle large volumes of data and can scale horizontally to handle increasing workloads and data volumes.

4. **Advanced Analysis:** They offer advanced tools and functionality for time series analysis, including forecasting algorithms, anomaly detection and trend analysis.

In summary, time series databases are essential tools for managing and analysing time series data, enabling organisations to derive value from these important sources of information in the modern data world.

## 2.2 Oracle Database

The Oracle Database, developed by Oracle Corporation, stands as one of the globe's preeminent relational database management systems (RDBMS). It is meticulously crafted to efficiently store and manage vast data sets, furnishing advanced functionalities for data organization, retrieval, analysis, and administration. Operating on a client-server architecture, Oracle Database entails at least two pivotal components: the database itself, constituting a structured collection of data housed in physical files, and the instance, encompassing memory areas and background processes essential for system functionality.

### 2.2.1 Database and Instance

While the database comprises the actual data and metadata, the instance acts as an interface enabling client applications to access the database. Notably, an instance must be associated with a physical database to enable data access, and while multiple instances can concurrently access the same database, each instance can only access one database at a time.

### 2.2.2 Data Storage and Architecture

Within the Oracle Database architecture, data is stored following both physical and logical schemas. The physical schema delineates the actual data storage structure, encompassing data files, control files, and online redo log files, while the logical schema facilitates granular disk space management through data blocks, extents, segments, and tablespaces. The instance, serving as the gateway for client connections, comprises the System Global Area (SGA), Program Global Area (PGA), and background processes, collectively ensuring seamless database operations.

For further insights and a visual representation of the Oracle Database architecture, reference Figure 2.3.

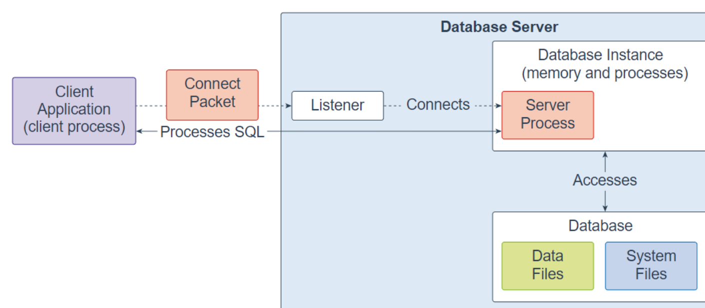


**Figure 2.1:** Oracle Database Architecture [1]

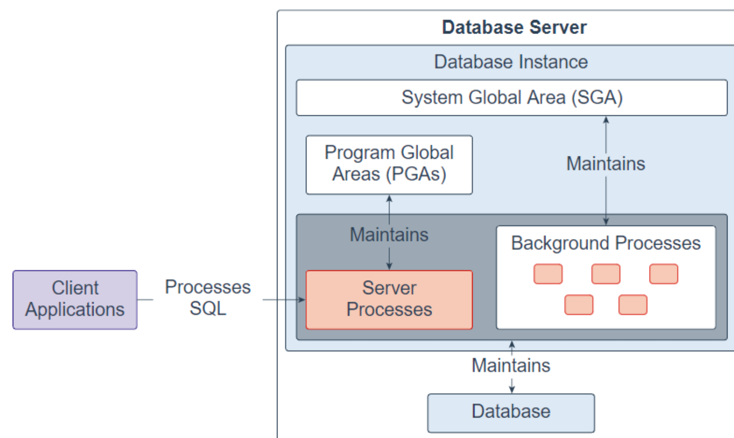


**Figure 2.2:** Oracle Database Instance Architecture [2]

## 2.3 Oracle Enterprise Manager (EM)

Oracle Enterprise Manager (EM) [3] is a database and application management platform provided by Oracle Corporation. It offers a comprehensive set of tools for monitoring, managing, and automating IT environments, enabling organizations to optimize performance, ensure availability, and manage IT service costs.

### 2.3.1 Monitoring Utilities

Oracle Enterprise Manager is particularly useful for monitoring systems, offering:

1. **Performance Monitoring:** EM continuously monitors database, application, and server performance, allowing administrators to view and analyze performance metrics in real time. This helps to quickly identify and resolve any performance issues that could affect business operations.

2. **Anomaly Detection:** EM uses advanced algorithms to automatically detect anomalies in systems performance, alerting administrators to any problems or abnormal patterns that require attention.

3. **Event Management:** EM collects and analyzes system events in real time, enabling administrators to monitor system activity and respond promptly to critical events or emergency situations.

4. **Reporting and Analysis:** EM provides tools to generate detailed reports on system performance, enabling administrators to analyze trends over time and identify areas for improvement.

### 2.3.2 Oracle Enterprise Manager Tables for Metrics

Within Oracle Enterprise Manager, you can find several tables that store metrics and monitoring information. Some of the most useful tables include:

1. **MGMT$ALERT_CURRENT:** This table displays current information for any alerts logged in the Management Repository that are in a non-clear state. It shows only the most recent open alert in a non-clear status for a given metric.

2. **MGMT$AVAILABILITY_CURRENT:** This table displays information about the most recent target availability information stored in the Management Repository.

3. **MGMT$METRIC_CURRENT:** This table displays information on the most recent metric values that have been loaded into the Management Repository.

These tables, along with others in Oracle Enterprise Manager, provide a wide range of useful data for performance monitoring and IT systems management.

## 2.4 Prometheus

Prometheus is a sophisticated open source tool designed for monitoring and analysing the performance of IT systems. Its origin dates back to SoundCloud in 2012, where it was developed to address the monitoring needs of a complex and growing cloud infrastructure. Since then, Prometheus has gained popularity and widespread adoption in the developer community due to its flexible architecture and numerous advanced features [4].

Prometheus collects a wide range of metrics that can be used to monitor and analyse system performance. Metrics are numerical indicators that represent specific aspects of system resources and services, such as CPU, memory,

network, disk utilisation and HTTP requests. Each metric is identified by a unique name and may include a series of labels that provide further context and information on the collected data.

The structure of metrics in Prometheus is hierarchical and organised to allow easy navigation and aggregation of data. Each combination of metrics and labels creates a unique timeseries. Labels are key-value pairs associated with a metric and allow data to be distinguished and grouped according to specific criteria. They can provide additional information such as the method of the request, the path to the requested resource and the status code of the response in an HTTP request tracking context.

Labels play a key role in enabling users to perform targeted queries and analyses on the monitoring data. They can be used to filter, aggregate and group data according to specific criteria, allowing operators to gain a detailed view of system performance and identify any problems or anomalies.

For example, using PromQL, it is possible to run queries that select only data related to certain labels or aggregate data according to certain labels. This gives users a high degree of flexibility and control in analysing monitoring data and enables them to obtain detailed information on system performance.

In summary, Prometheus is a powerful open source tool that offers a wide range of functionalities for monitoring and analysing the performance of IT systems. Thanks to its flexible architecture, powerful PromQL query language and efficient data management, Prometheus has established itself as one of the leading tools in the modern IT infrastructure monitoring landscape. It plays a key role in enabling operators to effectively monitor the performance of their systems and detect operational problems.

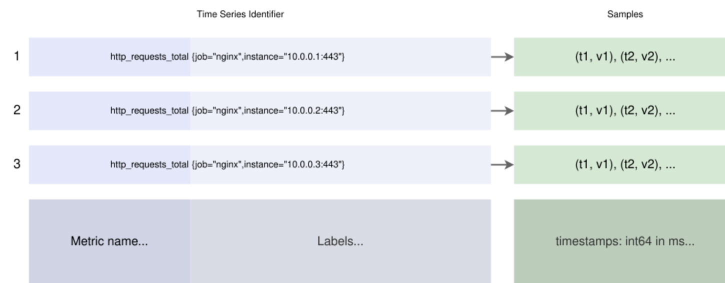


**Figure 2.3:** Prometheus Time Series structure [5]

### 2.4.1 Prometheus remote write

Prometheus remote write is a feature that allows Prometheus to send collected metrics data to a remote endpoint instead of storing it locally. This is particularly useful in distributed scenarios or where there are multiple instances of Prometheus, allowing monitoring data to be aggregated and stored in a single location.

In the context of our architecture, the use of Prometheus' remote write will prove invaluable in centralising and aggregating monitoring data from several distributed instances. This will provide a comprehensive and global view of system performance, facilitating the identification of trends and anomalies on a global scale.

In addition, sending monitoring data to a long-term storage database will allow us to retain historical data for future analysis and generate retrospective reports on system performance. This will help improve our ability to monitor performance trends over time and analyse any past operational problems.

Ultimately, the use of Prometheus' remote write will optimise the management of monitoring data on a global scale, allowing a more comprehensive view of system performance and providing useful information for the continuous improvement of operations.

## 2.5 Grafana

Grafana is a versatile and powerful platform for data visualisation and analysis, widely used in the monitoring and performance analysis of IT systems, but also in many other areas such as business data analysis and application observability. One of its most outstanding features is the ability to make certain dashboards standard, allowing users to easily share layouts, settings and views between different Grafana installations or with other team members. This facilitates greater consistency in data analysis and simplifies collaboration between teams, facilitating the sharing of best practices and collaborative problem solving. [6]

One of the most important aspects for which Grafana is known is its efficiency in loading and processing data, which allows users to visualise large amounts of information in real time without compromising system performance. This feature is particularly important in monitoring contexts, where it is essential to have rapid and continuous access to data in order to make informed decisions and respond promptly to any problems. Grafana's ability to effectively manage large volumes of data contributes significantly to its usefulness and adoptability in complex operational environments.

Although there are tools similar to Grafana (such as Kibana), its strong compatibility with Prometheus is a key advantage. Grafana offers a number of Prometheus-specific features, such as the ability to execute PromQL queries directly from dashboards and the visualisation of metrics and time charts in an intuitive and effective manner. This tight integration with Prometheus makes Grafana a natural choice for visualising data collected by Prometheus and allows users to take full advantage of the capabilities of both platforms.

In addition, Grafana is extremely flexible and offers numerous possibilities

for extension and integration with other platforms and monitoring systems. This allows users to customise and adapt Grafana to their specific needs, integrating it with alerting systems, data analysis tools and other data sources beyond Prometheus, such as InfluxDB, Elasticsearch and many others. The wide range of available functionalities and integrations makes Grafana a valuable resource for a variety of use cases and operational contexts. We will look in more detail at these integrations in the [Future Works] section.

Finally, the active and supportive community surrounding Grafana further contributes to its value as a data monitoring and analysis platform. Thanks to the large community of developers and users who constantly contribute to the improvement of the platform, Grafana is able to offer continuous support, quick bug fixes and constant development of new features and improvements. This provides Grafana users with a dynamic and constantly evolving environment, with access to a wide range of resources and support to address the challenges of monitoring and analysing data effectively and efficiently.

## 2.6 Mimir

Mimir is a fundamental tool for the architecture we adopt in our work, offering a complete and scalable solution for managing monitoring data and creating a consolidated view of system performance. Its compatibility with Grafana makes it particularly valuable, enabling seamless integration between data collection from Prometheus and visualisation through Grafana. [7]

What sets Mimir apart is its ability to facilitate the implementation of Prometheus' remote write. This functionality allows Prometheus to send collected metrics data to a remote endpoint instead of storing it locally, greatly

enhancing data management and analysis at scale and in distributed environments. Mimir acts as a bridge between Prometheus and the remote storage system, enabling reliable and efficient transmission of monitoring data and long-term archiving for future analysis.

In addition, Mimir offers a number of advanced features that further enhance the management of monitoring data. For example, data compression and deduplication help optimise storage space and reduce costs, while data replication ensures greater system reliability and availability.

Although there are other tools similar to Mimir on the market, such as Thanos, its native compatibility with Grafana makes it a preferred choice for many. This integration simplifies the creation of interactive dashboards and visualisation of monitoring data, allowing users to easily gain a comprehensive and detailed view of system performance. Ultimately, Mimir is an essential component of our architecture, providing the functionality required for efficient and scalable management of monitoring data and ensuring reliable and comprehensive monitoring of our IT infrastructure.

# Chapter 3

# Methodology

## 3.1 Architecture

In this chapter, we will explore the methodology adopted to implement and evaluate the client-server architecture, supported by the various tools discussed above. We will begin by analysing the context in which the idea for this architecture was born and how a Proof of Concept (PoC) was developed to simulate its operation.

### 3.1.1 General Architecture

The proposed architecture (Fig.3.1) is designed to simulate a distributed client-server environment, with an infrastructure composed of a single container and two separate subnets. This subdivision makes it possible to faithfully reflect the dynamics of a private network of the client and the company, with its interactions and security requirements.

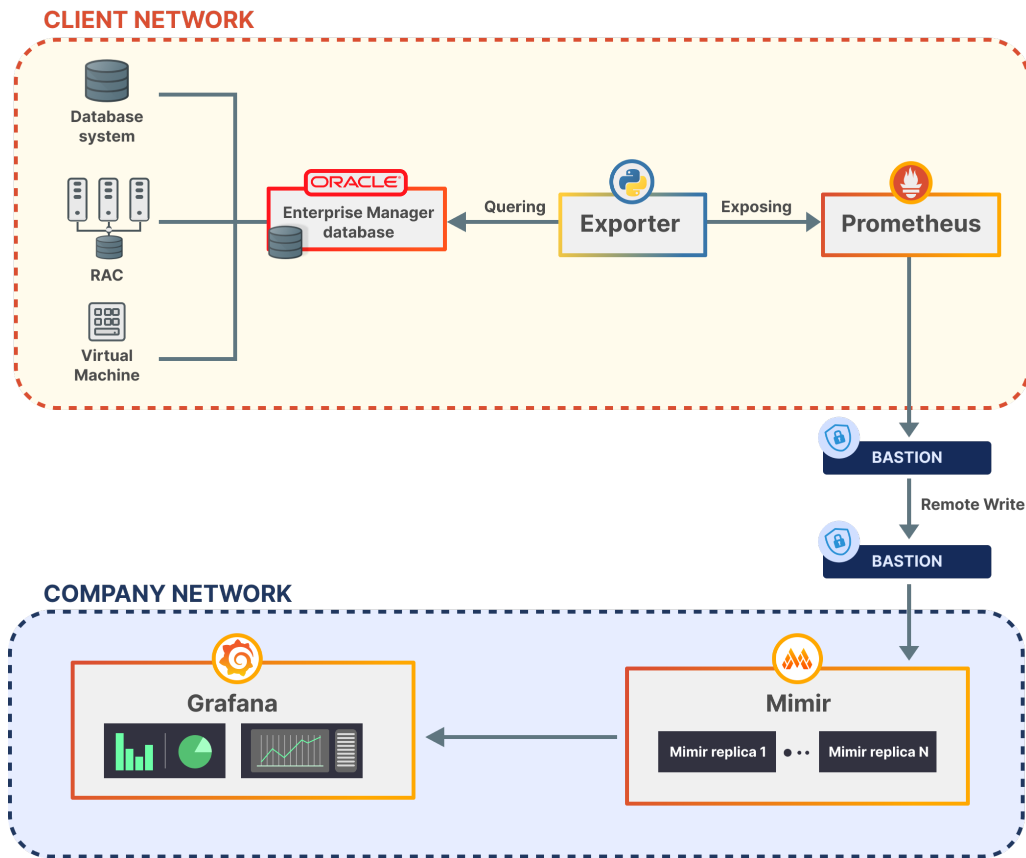


**Figure 3.1:** The architecture of proposed solution.

### 3.1.2 Subnet structure

Each subnet consists of two virtual machines (VMs): one to simulate the private network of the customer and the company, and a bastion to expose the public IP address for both networks. This setup reflects a typical network architecture, where the bastion acts as an external access point to the private network, providing an additional layer of security and access controls.

### 3.1.3 Role of the Container

The container plays a crucial role in the architecture, acting as a central point for the execution and management of the various services and applications. Within the container, the various tools, including Prometheus, Grafana, Mimir and others, are executed and orchestrated to enable the collection, storage, analysis and visualisation of monitoring data.

### 3.1.4 Architecture Objectives

The main objective of the architecture is to provide a flexible, scalable and secure environment for monitoring and analysing the performance of IT systems. The division into separate subnets makes it possible to isolate and manage customer and company networks separately, while ensuring secure and controlled communication between them. Furthermore, the architecture is designed to be easily scalable, allowing new resources and services to be added in accordance with the evolving needs of the organisation.

Through this architecture, we aim to provide a robust and reliable environment for monitoring data management, enabling operators to effectively monitor systems performance and respond promptly to any problems or anomalies.

### 3.1.5 Role of Prometheus and Mimir in the Architecture

Prometheus is installed in agent mode on the customer's private VM, configured in a lighter version with basic functionality, including remote write. This configuration allows Prometheus to collect and send monitoring data to the enterprise VM via remote write, ensuring efficient and scalable data

management on a large scale.

Once the data is received, Mimir takes care of the storage, ensuring that the data is stored in a reliable and accessible manner. Mimir manages the storage not only locally, but can also be configured to utilise remote storage solutions, enabling data to be stored on a large scale and for long periods of time. In the current PoC, the integration of remote storage is ignored.

### 3.1.6 Using Grafana for Data Visualisation

Grafana uses Mimir as a datasource to access archived monitoring data. Using PromQL, the Prometheus query language, Grafana allows complete manipulation of the data to visualise it in a way that best suits the business case. PromQL allows users to perform advanced queries and data analysis, including aggregation, filtering and temporal manipulation operations to obtain a detailed and customised view of system performance. With Grafana, users can create interactive dashboards and customised visualisations to effectively and intuitively monitor and analyse data.

## 3.2 Configuration

### 3.2.1 Problem of Access to the Oracle Database and Enterprise Manager

A major challenge in this project is access to the Oracle database and the Enterprise Manager (EM). This issue arises both for practical reasons, given the vast amount of data generated by these platforms, and for security issues, since access to certain Metric Extensions of the EM requires authentication

by administrators.

### 3.2.2 Solution: Python Exporter for Oracle DB

To overcome these challenges, a Node Exporter written from scratch in Python was developed. Although an Oracle DB Exporter exists in its version 1.0.0, I preferred to create a custom implementation in Python. This approach allows for complete customization of queries and security, allowing more control and flexibility in accessing monitoring data. We will see later the configuration in YAML of the exporter for both DB access and metrics query.

### 3.2.3 Custom Python Exporter for Oracle DB

The Python Exporter for Oracle DB is designed to connect to the Oracle database and retrieve specific data through custom queries. Using the `cx_Oracle` [8] module to connect to the database and execute queries, the exporter is able to collect a wide range of monitoring metrics, including database performance data, resource utilization, and system information.

Because of the flexibility offered by Python, it was possible to implement additional security measures, such as managing credentials securely via yaml configurations for access to the Enterprise Manager's Metric Extension. This ensures that only authorized users can access sensitive and critical system data, while ensuring the security and integrity of monitoring information. From a real-world perspective, it enables credential management directly on the client side, which then does not have to worry about sharing passwords and authentications with the enterprise.

**Algorithm 1** Connecting to Oracle Database and Fetching Metrics

```
1: Read configuration from a YAML file config.yml
2: Save authentication variables from configuration
3: Read queries and metric names from file default-metrics.yml
4: while true do
5:     Open a connection with Oracle Database
6:     for all queries in metrics do
7:         Execute the query
8:         Fetch the result
9:         Save the SELECT statement of the query as labels
10:        Add the values fetched along with their respective labels to the
   Prometheus metric
11:        Close the connection
12:        Depending on log level, write [INFO|ERROR|WARNING]
13:    end for
14:    Wait for a certain period of time
15: end while
```

### 3.2.4 The configuration file

The configuration file (`config.yml`) contains essential information for accessing the Oracle database, ensuring secure management of sensitive credentials. Among the main fields are:

- **Username**: This field specifies the username required for authentication to the Oracle database. You can configure it with the appropriate credentials for authorized access to the database.

- **Password**: This field specifies the password associated with the username for authentication to the Oracle database. It is critical to maintain the security of this information and to avoid sharing it in an insecure manner.

- **Connection String**: The connection string defines the path and details required to establish a connection to the Oracle database. This includes the database name, server IP address, port, and other system-specific configuration information.

- **ORACLE_HOME**: This environment variable specifies the path to the root Oracle installation directory on the system. It is essential for correctly locating the executable files and libraries required for proper operation of Oracle applications.

- **ORACLE_SID**: This environment variable specifies the system identifier (SID) of the Oracle database to which you want to connect. Each Oracle database running on the system will have a unique SID that uniquely identifies the database in the context of the operating system.

```yaml
auth_config:
  DB_USER: "dbsnmp"
  DB_PASSWORD: "dbsnmp"
  DB_DSN: "EMREPCDB"

env_config:
  ORACLE_HOME: "/u01/app/oracle/product/19.0.0/dbhome_1"
  ORACLE_SID: "emrepcdb"
```

**Figure 3.2:** A *config.yml* snapshot

In the next chapter we will see how this tools will interact between.

# Chapter 4

# Proposed Solution

In this chapter, we will explore the details of the steps, choices and motivations that guided our implementation. Furthermore, we will analyse how the tools mentioned in the previous chapter interact with each other to ensure effective and comprehensive monitoring of the system.

## 4.1 How Prometheus works and how Node Exporter interact with

First of all, we will elaborate on how Prometheus handles the structure of the collected data, organising it in a hierarchical manner.

Prometheus Data Structure Prometheus manages data using a key concept called Metrics. Each metric can be provided with various labels, which allow context and detail to be added to the measurements. It is important to note that each unique combination of labels within a metric creates a distinct timeseries.
Prometheus metrics can take many forms, ranging from basic metrics such as counters and meters, which provide information on trends and the state of the

system over time, to more complex metrics such as histograms and summaries, which allow you to analyse the distribution of data and identify any anomalies or performance issues. After examining the data structure of Prometheus, it is essential to understand the scraping process, i.e. the mechanism by which Prometheus collects data from the various monitoring sources.

### 4.1.1 How Prometheus Scraping Works

Prometheus uses a scraping model to collect data from its monitoring targets. This approach is based on a pull-type architecture, in which Prometheus itself makes periodic requests to the services it wishes to monitor, called 'targets'.

### 4.1.2 Target Selection

Target selection is done via the Prometheus configuration. Users can specify the desired targets in the Prometheus configuration file, indicating the address of the service to be monitored and other relevant information such as the path of the exposed endpoint and any security options. Once the targets are configured, Prometheus begins making regular requests to each target to collect monitoring data. This data is then processed and stored within Prometheus for analysis and visualisation.

In the example shown in Figure 4.1, we can see how the value of a label within a metric generates two separate timeseries, each of which scrapes the target at regular intervals.

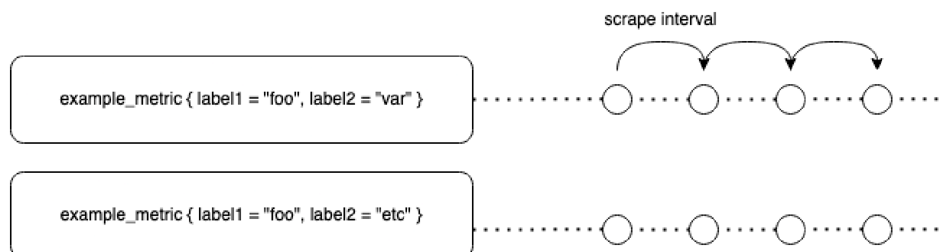


**Figure 4.1:** Example of different timeseries

### 4.1.3 Scraping process for Node Exporter

The exporter is the essential part for retrieving the monitoring data. As explained in the previous chapter, the node exporter connects to the database on the client's virtual machine using the credentials specified in the configuration file.

Once the connection is established, the node exporter retrieves the name of the metrics, the queries to be executed and their labels from a file called default-metrics.yml (as shown in the Figure 4.2). The configuration file also specifies the scraping interval and the HTTP port on which to write the data.

Next, Prometheus scrapes the exporter port to retrieve all available metrics. This solution allows you to have several exporters active at the same time, allowing you to differentiate between different monitoring operations on the same or different targets. Furthermore, new metrics can be added easily and flexibly by editing the default metrics file.

The queries used in the PoC exporters include a SELECT of the labels shown in the table below (4.1).

| LABEL NAME | DESCRIPTION |
|---|---|
| **target_type** | The type of target (i.e. Pluggable, Container or RAC database) |
| **target_name** | The name of the target |
| **metric_name** | The name of metric class (i.e. Load, Memory, CPU, MetricExtension) |
| **metric_column** | The name of the metric associated with class (i.e. 'UsedPct' for Memory) |
| **line_of_bus** | Name of the customer |
| **lifecycle** | The group of target (i.e. 'Production', 'MissionCritical', 'Test') |
| **key_val** | Typically additional information about metrics |
| **availability_status_code** | Info about target's availability status |

**Table 4.1:** Labels extracted with SELECT in metrics' query

```
- name: 'exporter_metric_availability'
- description: 'Availability of the targets'
- type: gauge
- query: "SELECT
  m.TARGET_NAME,
  t1.property_value as LINE_OF_BUS,
  t2.property_value as LIFECYCLE,
  m.TARGET_TYPE,
  m.AVAILABILITY_STATUS_CODE
FROM
  mgmt_view.MGMT$AVAILABILITY_CURRENT m
LEFT JOIN (
    select TARGET_GUID, property_value
    from mgmt_view.cm$em_tprops_ecm_view
    where property_name = 'orcl_gtp_line_of_bus'
  ) t1
  ON m.TARGET_GUID = t1.TARGET_GUID
LEFT JOIN (
    select TARGET_GUID, property_value
    from mgmt_view.cm$em_tprops_ecm_view
    where property_name = 'orcl_gtp_lifecycle_status'
  ) t2
  ON m.TARGET_GUID = t2.TARGET_GUID"
- labels:
  - label_name: TARGET_NAME
  - label_name: LINE_OF_BUS
  - label_name: LIFECYCLE
  - label_name: TARGET_TYPE
  - label_name: AVAILABILITY_STATUS_CODE
```

**Figure 4.2:** An example of metrics configuration for monitoring

## 4.2 Remote Write and Mimir

Once Prometheus has retrieved the data via the node exporter, it proceeds to push for Remote Write. In this PoC, we did not focus on security for sending packets between VMs, so for ease of use we only allowed pushing on certain ports.

In the Prometheus configuration, we entered the IP address of the company's VM-bastion, on which Mimir and Grafana are installed. Using the Prometheus API, metrics are sent to Mimir's listening port.

This approach allows Prometheus to efficiently send the collected data to Mimir for storage and subsequent analysis, without having to store the data

locally and ensuring centralised and scalable management of the metrics. Configuring the IP address and port where Mimir is listening allows Prometheus to reliably send data to its destination, enabling Mimir to efficiently and securely manage and analyse monitoring data.

### 4.2.1 Mimir object storage and ring configuration

When configuring Mimir to receive the remote write from a Prometheus on a different VM, several variables were considered to ensure optimal operation and effective management of monitoring data.

#### Configuring the Remote Write

To configure Mimir for remote write from a Prometheus on a different VM, the listening interface and port on which Mimir accepts remote write requests was specified. This allows Prometheus to send monitoring data to Mimir via a secure and reliable connection.

#### Ring

Mimir uses a distributed data structure called a 'ring' to store and manage monitoring data. The ring is a node structure that allows data to be distributed over multiple nodes and offers greater scalability and reliability than a single node. In addition, the ring provides fault tolerance, allowing Mimir to maintain data accessibility even if one or more nodes fail.

#### Local Storage

In addition to the distributed ring, Mimir also uses local storage to temporarily store monitoring data before it is distributed in the ring. Local storage allows for fast writing and reading of data and offers greater resilience in the event of connectivity problems or node failures. Once the data has been written to the local storage, it is then distributed in the ring for long-term persistence and analysis.

#### Other Details

In addition to the configuration of the remote write, ring and local storage, other configurations and optimisations were considered to ensure the performance and stability of Mimir. These may include data replication configuration, capacity management, backup configuration and data retention policy management.

Overall, the configuration of Mimir for remote write from a Prometheus on a different VM was designed to provide a robust and scalable solution for storing and analysing monitoring data, enabling users to obtain useful and meaningful information on system performance.

## 4.3 Data Manipulation and Visualization with Grafana

After configuring and populating Mimir with monitoring data from Prometheus, the next step is to connect Mimir to Grafana to create meaningful and informative visualisations of the data. This section will explore the process of configuring data sources on Grafana and manipulating data using PromQL to

create effective visualisations of monitoring metrics.

### 4.3.1 Data Source

To enable Grafana to display data from Prometheus and the node exporter, it is necessary to configure the data sources. Via the administration panel, we can connect Grafana to Mimir, which allows us to access Prometheus without having to know the IP address of the VM from which the data comes (Fig. 4.3).

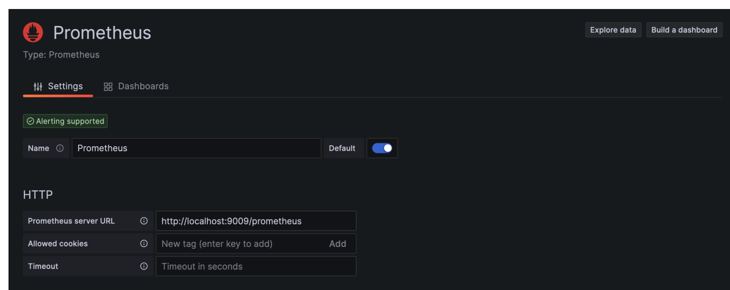


**Figure 4.3:** Data-source on Grafana

At this point, we are ready to proceed with the manipulation of the time series.

### 4.3.2 PromQL and metrics

As discussed in previous chapters, Prometheus organises time series within a data structure called a metric. Metrics can contain a wide range of different time series, and to select the data of interest, we use **PromQL**, a powerful and flexible query language that allows us to perform complex operations on the collected data.

Using the extracted labels (see Table 4.1), we can easily derive the time series of our interest. Similar to the WHERE clauses in SQL, we can set conditions for the labels to filter the data. For example, the label 'line_of_business' allows us to easily select the desired customer.

The following figure shows an example of monitoring the CPU utilisation percentage for all hosts on the Mediamente server. [4.4]

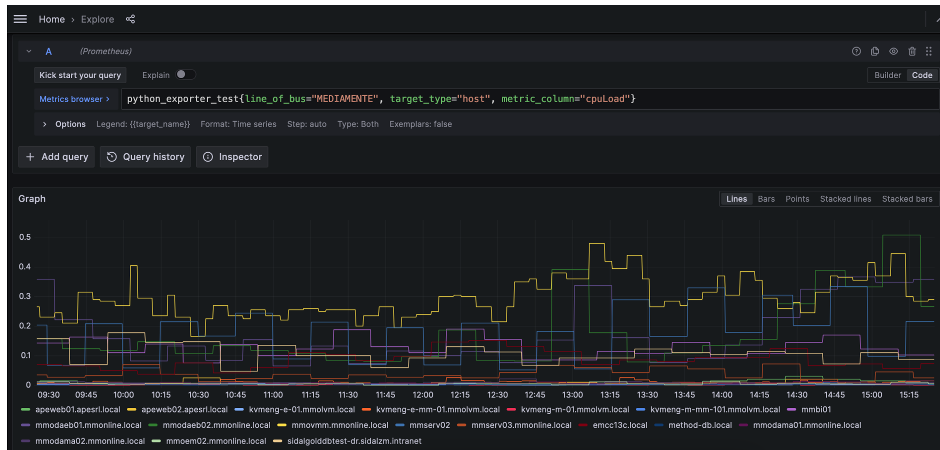


**Figure 4.4:** Example of cpu% timeseries

### 4.3.3 Dashboard and global variables

Grafana uses the concept of a Dashboard as a fundamental tool for organizing and displaying multiple visualizations in a single interface. A Dashboard can contain a series of panels, each representing a specific data visualization. This approach is extremely useful because it allows users to group related visualizations into a single screen, making it easier to analyze and understand the data.

One of the most powerful features of Dashboards in Grafana is the ability to use Global Variables. Global Variables are dynamic variables that can be used within queries and panel settings to allow dynamic interaction with the data. This is particularly useful when you want to create visualizations that depend on specific parameters or filter conditions.

In the context of our project, the use of Dashboards and Global Variables

allows us to group certain related queries and visualization settings within a single Dashboard. For example, we can create a Dashboard specifically for monitoring hosts, which includes a series of panels showing CPU, memory, and disk usage metrics for each host. In addition, we can create another Dashboard dedicated to monitoring tablespaces, which displays disk space usage metrics for various database tables.

The use of Dashboards and Global Variables thus allows us to effectively organize and visualize monitoring information, enabling users to explore and analyze data more efficiently and intuitively. This approach contributes greatly to the creation of a comprehensive and highly functional monitoring system that meets the specific needs of our project.

Some examples of the work are showned in following figures [Fig. 4.5, 4.6, 4.7]

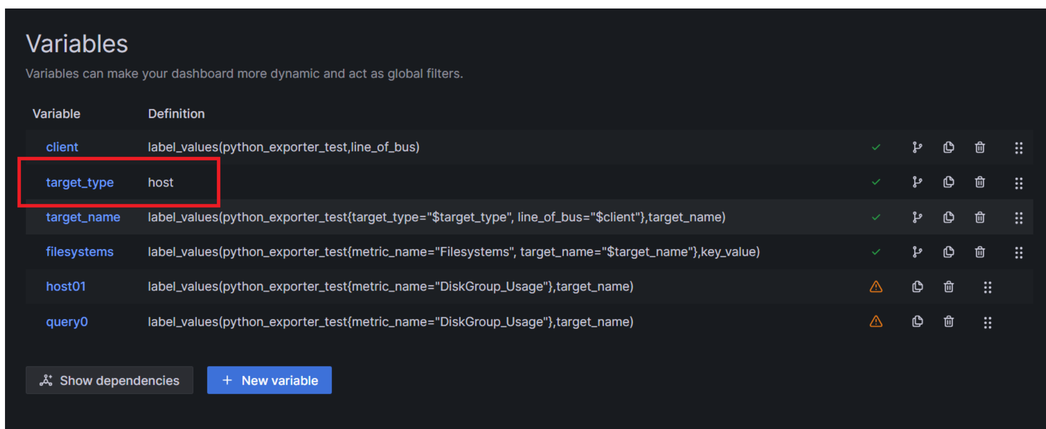


**Figure 4.5:** Global Variables for 'Host' dashboard.

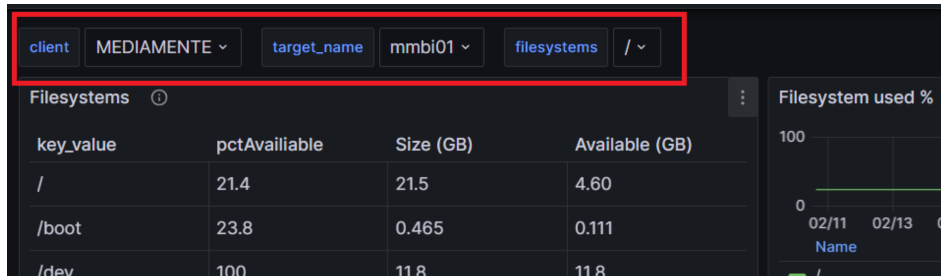


Figure 4.6: Example of global variables.

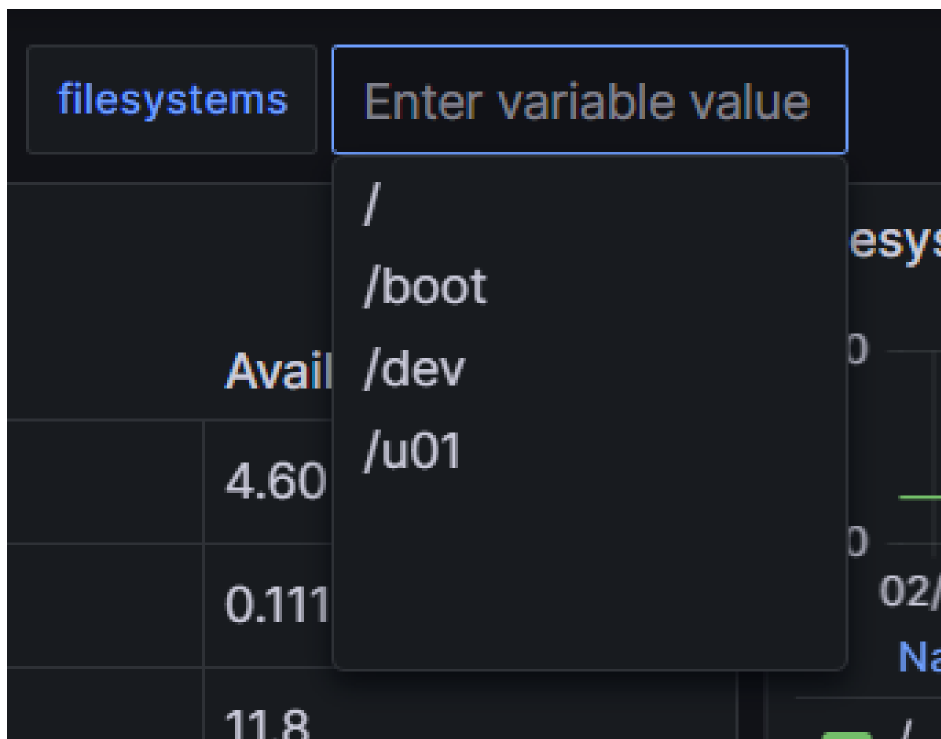


Figure 4.7: Variables selection on dashboard.

# Chapter 5

# Visualization and Results

## 5.1 Visualization

We will now explore the types of views used to monitor key metrics related to Filesystems and Tablespaces, which play a crucial role in the context of database monitoring.

Tablespaces are storage spaces within Oracle databases where data is stored. It is essential to constantly monitor the available space within tablespaces to avoid capacity problems and ensure efficient database operation. Total occupancy of disk storage, where instances reside, can cause significant slowdowns or even temporary disruptions. Constant monitoring is therefore a key tool for planning scaling and optimization operations.

The most commonly used views include TimeSeries, which allow you to view the history of parameters such as %Used, %Free, %Cpu, and so on, and Table, which provide an overview of monitoring, including Filesystems and disks. Using these views, trends and anomalies in the monitoring data can be easily identified, enabling timely interventions to ensure proper system

operation.

Below are some snapshots of the dashboard (Fig. 5.1, 5.2, 5.3).

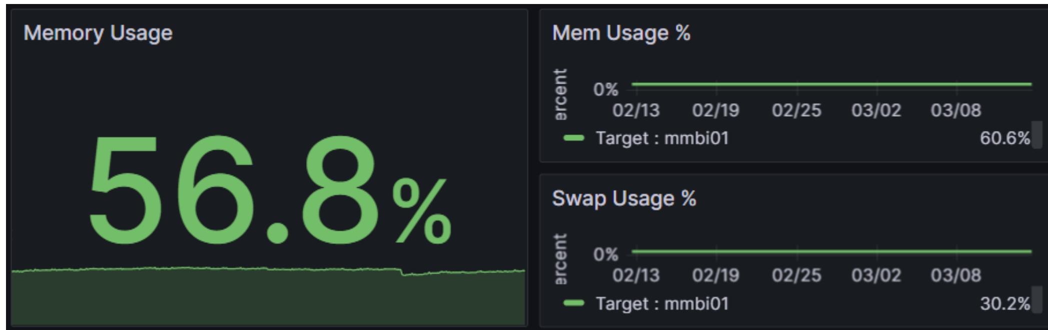


**Figure 5.1:** Timeseries and stats visualization example.

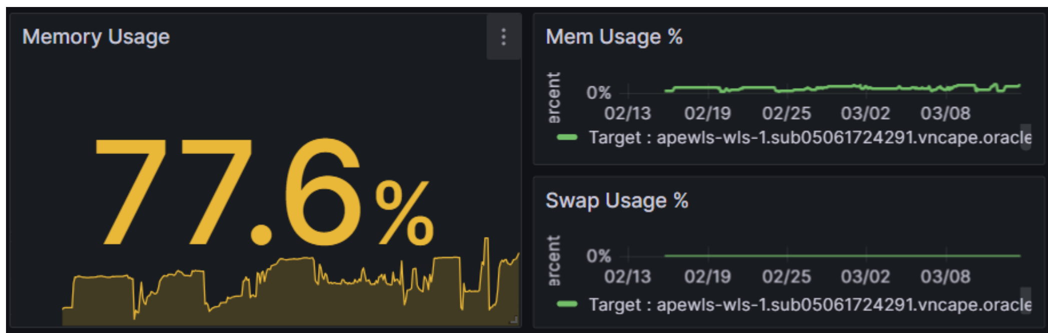


**Figure 5.2:** Different colors can be used to alert a critical threshold.

ME$REPORT_Tablespaces

| TABLESPACE_NAME | USED_PERCENT | CURR_USED_PCT | CURR_SIZE_GB | CURR_USED_GB | CURR_FREE_GB | DATAFILES |
|---|---|---|---|---|---|---|
| ALERTPA | 2.13 | 46.7 | 2.92 | 1.36 | 1.55 | 1 |
| ALERTPA_DOC | 4.06 | 86.8 | 8.21 | 7.13 | 1.08 | 1 |
| APEX | 0.760 | 90.3 | 1.48 | 1.34 | 0.144 | 1 |
| APEX_1270312578... | 69.8 | 69.8 | 0.0977 | 0.0681 | 0.0295 | 1 |
| APEX_1798600915... | 2 | 66.4 | 0.0147 | 0.00980 | 0.00490 | 1 |
| FLOW_240702368... | 0.940 | 0.944 | 0.977 | 0.00920 | 0.967 | 1 |
| GGS | 0.790 | 33.2 | 1.52 | 0.506 | 1.02 | 1 |

**Figure 5.3:** Table visualization example.

## 5.2 Results

We will now explore the final dashboards of this project, examining the choices that guided the use of specific metrics and the visualization methods adopted.

The dedicated Hosts dashboard presents a number of visualizations crucial for quick and effective system analysis. Tables and time series showing trends in available space on different mount points are of critical importance in disk space management.
In addition, the panels devoted to CPU and RAM utilization provide essential

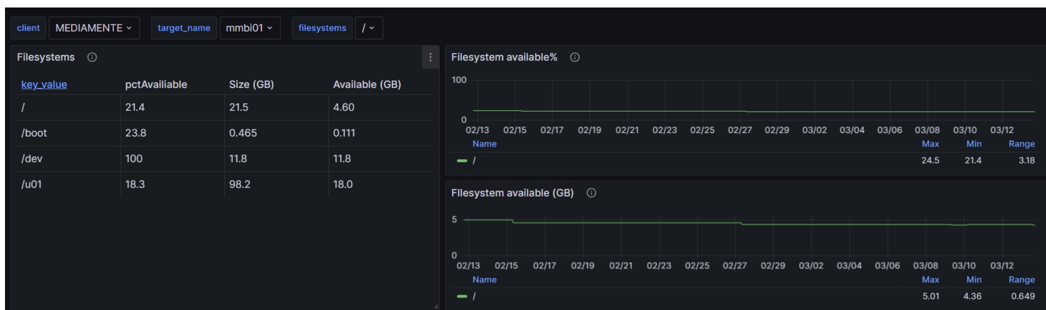


**Figure 5.4:** Filesystems monitoring panels.

information on the health of the target, allowing any spikes or anomalies in system performance to be quickly identified (Fig. 5.5).
Finally, the Disk Group utilization table provides a detailed analysis of the storage resources used (Fig. 5.6).

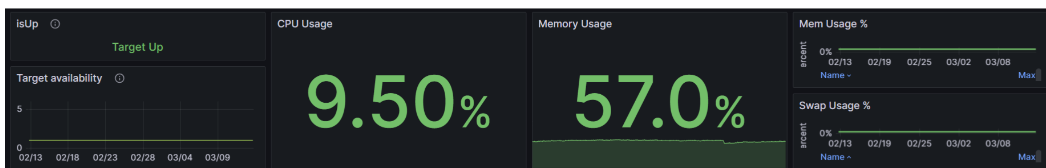


**Figure 5.5:** Panels for health's device monitoring.

In the dashboard dedicated to databases, we find a series of tables useful

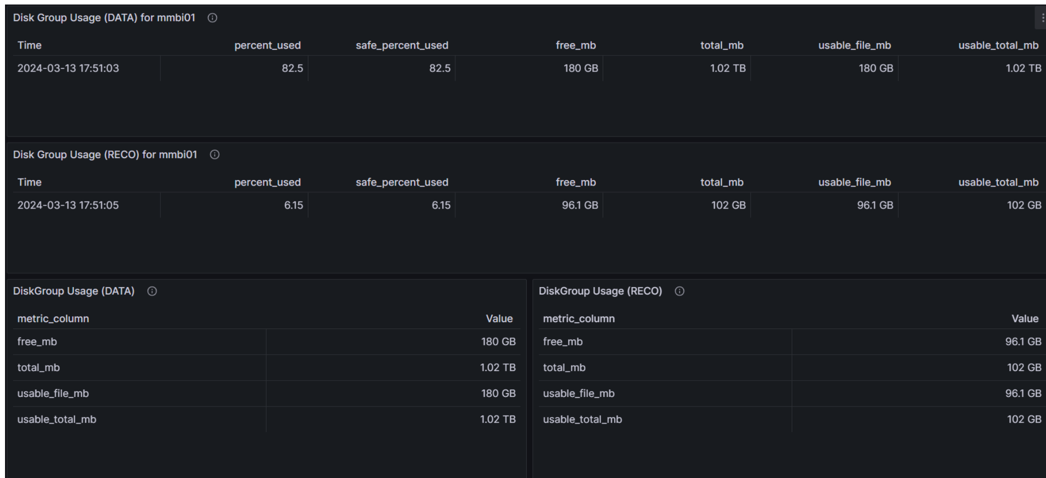


**Figure 5.6:** Disk Group Usage info.

for visualizing the tablespaces present on each target and their utilization in percent and gigabytes of storage.

Time-series views of these values provide an in-depth analysis of the memory usage trend and database fill rate, enabling optimal management of available space (Fig. 5.7).

A crucial element of this dashboard are the tables that associate the Datafile

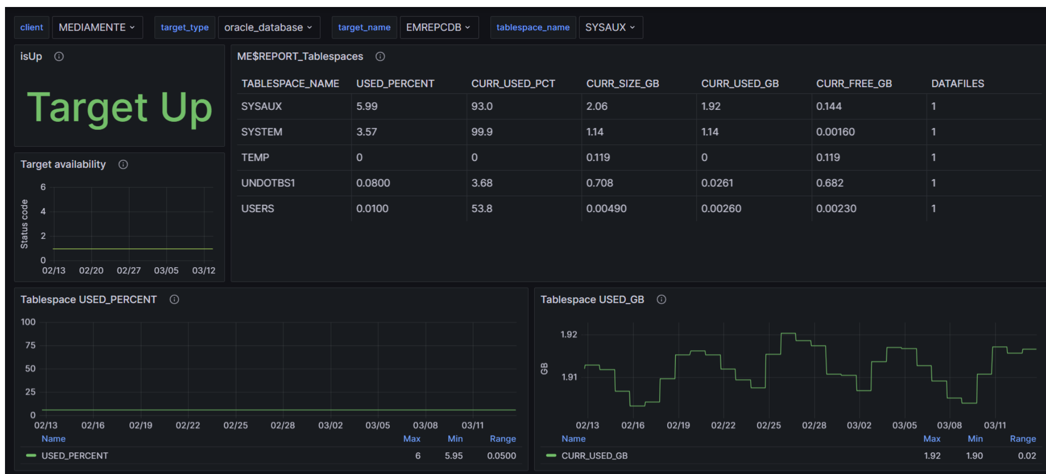


**Figure 5.7:** Tablespaces info.

Path with each tablespace, allowing direct identification of the disk on which

the data is stored and facilitating troubleshooting or operational needs (Fig. 5.8).

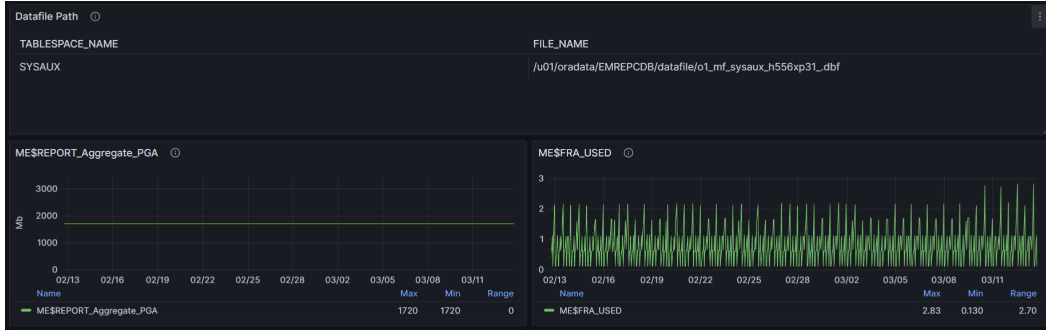


**Figure 5.8:** Datafiles path info.

# Chapter 6

# Conclusions

In this concluding chapter, we will examine the key elements that made possible the success of the solution adopted for multi-target monitoring. During the development of this project, we adopted a highly efficient and versatile architecture that proved extremely suitable for our multi-target monitoring needs.

A key element of this architecture was the Node Exporter, a highly customizable component that allowed us to collect a wide range of metrics from our systems, adapting perfectly to the specific needs of each client. Its versatility allowed us to obtain detailed and reliable data on CPU, memory, network and more, without compromising system resources.

In addition, using Prometheus with the Remote Write feature allowed us to centralize the storage of monitoring data on a single infrastructure, minimizing the deployment and management of monitoring servers. This greatly simplified the data management and analysis process, ensuring an efficient and secure flow of information between the various system components.

The dashboards created in Grafana have played a key role in providing

effective and immediate visualization of the data of greatest interest to our clients. Thanks to their intuitive design and customization capabilities, we were able to provide our clients with a tailored monitoring experience tailored to their specific needs.

## 6.1 Future Works

Looking to the future, the design provides a solid foundation on which to build further enhancements and expansions. The architecture was designed with flexibility in mind, enabling dashboard additions and changes to be made quickly and easily to meet specific customer needs. This adaptability will ensure that the monitoring system can continue to grow and evolve with the changing needs of the company and its customers over time.
In particular, it is planned to implement a critical event alert system, integrated with the Prometheus AlertManager. This will enable immediate notifications in case of anomalies or major problems in the monitored systems. In addition, it is planned to further enhance this alerting system using GrafanaOnCall, which will allow emergency calls to employees to be handled in a timely and effective manner, ensuring immediate response to critical monitoring situations.

# Appendix A

# Node Exporter

```
import argparse
import os
import yaml
import cx_Oracle
import time
import prometheus_client
from prometheus_client import start_http_server, Gauge, Counter,
    Histogram, CollectorRegistry
import logging


def read_yaml_config(file_path):
    try:
        with open(file_path, 'r') as config_file:
            config_data = yaml.safe_load(config_file)

        # Check if the required variables are present in the
    configuration data.
        # In this example, only 'auth_config' and 'env_config'
    are checked
```

```
18          auth_variables = ['DB_USER', 'DB_PASSWORD', 'DB_DSN']
19          env_variables = ['ORACLE_HOME', 'ORACLE_SID']
20          for variable in auth_variables:
21              if variable not in config_data['auth_config']:
22                  raise KeyError(f"Required variable '{variable}'
    is missing in the configuration file.")
23
24          for variable in env_variables:
25              if variable not in config_data['env_config']:
26                  raise KeyError(f"Required variable '{variable}'
    is missing in the configuration file.")
27
28          return config_data
29      except FileNotFoundError:
30          print(f"File not found: {file_path}")
31      except KeyError as e:
32          print(f"Configuration error: {str(e)}")
33      except Exception as e:
34          print(f"An error occurred while reading the YAML file: {
    str(e)}")
35
36
37
38  def get_config_file_path():
39      # Check if the --config-file argument is provided.
40      parser = argparse.ArgumentParser(description="Read a YAML
    configuration file")
41      parser.add_argument("--config-file", help="Path to the YAML
    configuration file")
42      args = parser.parse_args()
43
44      # If --config-file argument is provided, use it.
45      if args.config_file:
46          return args.config_file
47      else:
```

```
48          raise FileNotFoundError()
49
50
51
52  # Eventually add other variables, set environment variables
     ORACLE_HOME and ORACLE_SID
53  def initialize_db_variables(config_data):
54      db_user = config_data['auth_config']['DB_USER']
55      db_password = config_data['auth_config']['DB_PASSWORD']
56      db_dsn = config_data['auth_config']['DB_DSN']
57      try:
58          os.environ['ORACLE_HOME'] = config_data['env_config']['
     ORACLE_HOME']
59          os.environ['ORACLE_SID'] = config_data['env_config']['
     ORACLE_SID']
60      except Exception as e:
61          print(f'Error setting ORACLE_HOME and/or ORACLE_SID: {str
     (e)}')
62      return db_user, db_password, db_dsn
63
64  # Return two dictionaries { metric_name : gauge/counter/histo } ;
      { metric_name : query } with all metrics and requests
     specified in default-metircs.yml
65  def initialize_prometheus_metrics(config_file):
66      with open(config_file, 'r') as stream:
67          config = yaml.safe_load(stream)
68      if stream is not None:
69          print("file_exist!")
70      metrics = {}
71      requests = {}
72      # Retrieve information about metric and create a new metric
73      for metric_config in config.get('metrics', []):
74          name = metric_config['name']
75          description = metric_config['description']
76          metric_type = metric_config['type']
```

```
77          labels = metric_config.get('labels', [])
78          query = metric_config['query']
79
80          if metric_type == 'gauge':
81              metrics[name] = Gauge(name, description, labelnames=[
    label['label_name'] for label in labels])
82              requests[name] = query
83          elif metric_type == 'counter':
84              metrics[name] = Counter(name, description, labelnames
    =[label['label_name'] for label in labels])
85              requests[name] = query
86          elif metric_type == 'histogram':
87              metrics[name] = Histogram(name, description,
    labelnames=[label['label_name'] for label in labels])
88              requests[name] = query
89      return metrics, requests
90
91
92
93  def collect_metrics(metric_query, config_data, logger, isUp):
94      # metric_query is a tuple (metric, query)
95      db_user, db_password, db_dsn = initialize_db_variables(
    config_data)
96      try:
97          connection = cx_Oracle.connect(db_user, db_password,
    db_dsn)
98          logger.debug(f"Connection established.\n{config_data['
    auth_config']}")
99          cursor = connection.cursor()
100         cursor.execute(metric_query[1])
101         rows = cursor.fetchall()
102
103         labels = metric_query[0]._labelnames
104         logger.debug("Query executed.") # Commented for testing.
    Low scrape interval.
```

```
105         for row in rows:
106 
107             # We store in a dict the value of each label {
    label_name : label } and last value of 'row' is the actual
    value
108             labels_dict = {labels[i]:row[i] for i in range(len(
    labels))}
109             value = row[-1]
110             if value is None:
111                 logger.debug("Found a None value.")
112                 continue
113             metric_query[0].labels(**labels_dict).set(value)
114         cursor.close()
115         connection.close()
116         logger.debug("Connection closed.")
117         host = os.environ['HOSTNAME']
118         SID = config_data['env_config']['ORACLE_SID']
119 
120         isUp.labels(host=host, SID=SID).set(1)
121     except cx_Oracle.DatabaseError as e:
122         logger.error(f"Database error: {e}")
123         isUp.labels(host=host, SID=SID).set(0)
124     except Exception as e:
125         logger.error(f"An error occurred: {e}")
126         isUp.labels(host=host, SID=SID).set(-1)
127 
128 def main():
129     # Initialize variable from config.yml and create a logger '
    python_exporter.log'
130 
131     # Get the directory path of the script
132     path_dir = os.path.dirname(os.path.abspath('__file__'))
133 
134     config_file_path = get_config_file_path()
135     config_file_path = path_dir+'/'+config_file_path
```

```
136 
137     # Interrupt the code if config.yml is not available (not
    given by command line or not present in $PATH)
138     if config_file_path is not None:
139         config_data = read_yaml_config(config_file_path)
140         if config_data:
141             print("Configuration data loaded.")
142         else:
143             print("No configuration data loaded.")
144             return
145 
146     # From here we start to collect the metrics
147     # NOTE: gauge_name, gauge_desc, labels, port and time
    duration in second for sleep can be set by config.yml (in
    future)
148     #gauge_name = 'python_test_exporter'
149     #gauge_desc = 'Test exporter description.'
150     #labels = ['target_name', 'target_type', 'metric_name', '
    metric_column', 'kv1', 'kv2', 'kv3', 'kv4', 'kv5', 'kv6', 'kv7
    ']
151 
152     port = config_data['scrape_params']['port']
153     scrape_interval = config_data['scrape_params']['
    scrape_interval']
154 
155     filename = config_data['logger']['logger_filename']
156     logger_level = config_data['logger']['logger_level']
157 
158     logging.basicConfig(filename=f'{path_dir}/{filename}.log',
    level=logger_level, format='%(asctime)s - %(levelname)s: %(
    message)s')
159     logger = logging.getLogger(f'{filename}')
160 
161     # Create a Prometheus gauge for isUp
162     #host = os.environ['HOSTNAME']
```

```
163     #SID = config_data['env_config']['ORACLE_SID']
164     labels_isUp = ['host', 'SID']
165     isUp = Gauge('isUp', 'Check if connection is established. ###
         0 : Dead | 1 : Alive | -1 : Unkown Status', labels_isUp)
166 
167     # Start the Prometheus HTTP server and initialize
        metrics_gauge
168     start_http_server(port=port)
169     metrics, requests = initialize_prometheus_metrics(f'{path_dir
        }/default-metrics.yml')
170     while True:
171         for metric_name, query in requests.items():
172             if metric_name in metrics:
173                 metric = metrics[metric_name]
174                 collect_metrics((metric,query), config_data,
        logger, isUp=isUp)
175         time.sleep(scrape_interval*60)  # scrape_interval is in
        minutes, check config.yml
176 
177 
178 
179 
180 if __name__ == "__main__":
181     main()
```

# Bibliography


[1] Oracle. *Oracle Database Architecture*. Accessed: February 2024. 2024. URL: https://www.oracletutorial.com/oracle-administration/oracle-database-architecture/ (cit. on p. 10).

[2] Oracle. *Oracle Database Instance Architecture*. Accessed: February 2024. 2024. URL: https://www.oracletutorial.com/oracle-administration/oracle-database-architecture/ (cit. on p. 11).

[3] Oracle. *Oracle Enterprise Manager*. Accessed: February 2024. 2024. URL: https://www.oracle.com/it/enterprise-manager/ (cit. on p. 12).

[4] Prometheus. *Prometheus - Monitoring system & time series database*. Accessed: February 2024. 2024. URL: https://prometheus.io/ (cit. on p. 13).

[5] Prometheus. *Prometheus TimeSeries structure*. Accessed: February 2024. 2024. URL: https://training.promlabs.com/training/introduction-to-prometheus/prometheus-an-overview/time-series-data-model (cit. on p. 15).

[6] Grafana Labs. *Grafana Open Source documentation*. Accessed: February 2024. 2024. URL: https://grafana.com/docs/grafana/latest/ (cit. on p. 17).

[7] Grafana Labs. *Mimir Overview*. Accessed: February 2024. 2024. URL: https://grafana.com/oss/mimir/ (cit. on p. 18).

[8] Oracle. *Python Cx Oracle Module*. Accessed: February 2024. 2024. URL: `https://pypi.org/project/cx-Oracle/` (cit. on p. 25).