SUP'COM

المدرسة العليا للمواصلات بتونس
ÉCOLE SUPÉRIEURE DES COMMUNICATIONS DE TUNIS

Telecommunications Engineer Cycle

Option :

SNI

End of Studies Project Report

Topic :

« Design and Development of a monitoring and log management solution »

Realised by :

Ameni ELHASSEN

Supervisors :
Saifeddine Berrayana ,Department Manager (Sofrecom)
Ahmed Belli,Technical Referent (Sofrecom)
Slim Rekhis (SUP'COM)

Work proposed and carried out in collaboration with:

Sofrecom
The Know-How Network

University Year : 2021/2022

Dedications

“

I dedicate this precious work to everyone who believed in me from day one :
To my father, ALi, who have always loved me unconditionally , supported and taught me to work hard for the things that I aspire to achieve.

To my mother, Jamila, for her endless love and affection ,thank you for all your supports, prayers, and advice, thank you for being patient and for believing in me.

To my brothers Amine and Walid , I can never imagine how would the world be without you. Thank you for always being there for me .
To my nieces,Lin and Loujayn, thank you for bringing the joy and light to my world.

To all my friends ,thank you for cheering me up in hard times and thank you for all your words of encouragement and prayers. To all my dear ones, to all of you Thank you.

”

ACKNOWLEDGMENTS

The accomplishment of this internship is the result of the contribution and the effective guidance of countless individuals who helped me during this project.

First of all,I want to express my deepest gratitude to the company in which I had my internship for the rich experience and the opportunity I had been offered.A special thanks to Mr. **Saifeddine Berrayana** and Mr.**Ahmed Belli** for their continuous support, daily supervision and wise words during this internship. The success of this internship is all thanks to their will to push me to make the best out of this project.

Sincere gratitude goes to Mr.**Slim Rekhis**, my academic tutor, for his availability , his valuable advice and words of encouragement.

I would also like to thank for the members of the **jury** for taking the time to evaluate my work and give me valuable feedback.

Finally, many thanks to all the people that had a direct or indirect contribution to the project and were not explicitly mentioned. Your help and support is very much appreciated.

Abstract

This report covers the implementation of a centralized monitoring and log management system for the detection of problems and anomalies in order to reduce error and incident rate . Several technologies were used to establish a set of dashboards to monitor the system's activity.
This project is carried out as part of an end-of-study internship in order to obtain the engineering degree at Sup'Com : Higher School of Communication of Tunis.

Key words : Prometheus ,Grafana ,Loki, Gitlab-Ci ,Ansible,Linux Administration, Bash scripting

Résumé

Ce rapport couvre la mise en place d'un système centralisé de surveillance et de gestion des journaux pour la détection des problèmes et des anomalies afin de réduire le taux d'erreurs et d'incidents. Plusieurs technologies ont été utilisées pour établir un ensemble de tableaux de bord permettant de surveiller l'activité du système.

Mots clés : Prometheus ,Grafana ,Loki, Gitlab-Ci ,Ansible ,Administration Linux ,scripting Bash ,

Contents

ListofFigures

ListofTables

Glossary of Acronyms

API	*Application Programming Interface*
DB	*Database*
HTTP	*Hypertext Transfer Protocol*
IT	*Information Technology*
ICMP	*Internet Control Message Protocol*
ODE	*Orange Deployment Engine*
TLS	*Transport Layer Security*
TSDB	*Time Series DataBase*
SMTP	*Simple Mail Transfer Protocol*
SNMP	*Simple Network Management Protocol*
SSH	*Secure Shell Protocol*
VM(s)	*Virtual Machine(s)*
VIO	*Vente Indirecte aux Opérateus*

Signatures

Sofrecom supervisor:

Sup'Com supervisor

General introduction

Companies today manage a wide number of computer systems that help them orchestrate processes and boost their efficiency. However, these systems continue to be subject of incidents, failures and various technical issues that may be of minor, major or critical nature.
With that being said, monitoring various platforms and exploiting the log files provided by different servers is a critical and fundamental point to ensure the sustainability and effectiveness of these systems and minimize operating losses, thus, ensuring that clients do not notice the occurrence of these events and therefore preventing their dissatisfaction.
On the other hand, monitoring optimizes decision-making and guarantees minimal intervention time. Therefore, companies must invest and change their strategies using data generated by servers to ensure stability, security, performance, and high availability of their information systems.
Hence, the goal of this end-of-study project is to implement a centralized, real-time monitoring and log management solution capable of collecting and querying metrics related to system, application, or logs. The solution should also ensure implementing different alerting techniques and automating the insertion of new targets to be monitored. And finally displaying different dashboards, to provide the observability throughout the entire infrastructure.
Our report is structured around three chapters :

- In the first chapter, where we present the host organization, the general context of the project, the problematic driving the implementation of this project and finally the key concepts of the project and a comparative study of the tools we used to achieve our objectives.
- The second chapter presents an overview of the solution and a detailed design of the project.
- The third chapter describes the hardware and software environment of the project, and focuses on the implementation stages of the solution.

Chapter 1

General context and concepts

Introduction

This chapter will briefly present the host the company in which I have conducted my end-of-study project, followed by a description of the needs driving the development of this solution as well as the key concepts applied throughout this project.

1.1 The host enterprise: Sofrecom

In this section, we will present the host company and its activity sector.

1.1.1 Enterprise presentation

Sofrecom is a subsidiary of Orange Group: it has developed 50 years of know-how in the operator sector, making it a global leader in consulting and telecommunications technology. The group accompanies its clients (operators, industrial companies, governments, etc.)in its projects management, creation, or recovery, depending on the sector of activity. Today, the Sofrecom Group has more than 2,000 consultants and specialists in 11 locations around the world, serving more than 200 clients in more than 100 countries through its subsidiaries and branches on 4 continents. Sofrecom Tunisia is a major actor in consulting and engineering in telecommunications on the local market.[1]

Sofrecom Tunisia , is the youngest subsidiary of the group, was initiated in October 2011, has today more than 950 engineers and experts. Sofrecom Tunisia has gained its strength and credibility through hundreds of international projects, putting it one step ahead of its competitors

Figure 1.1 describes the geographical distribution of Sofrecom around the world.

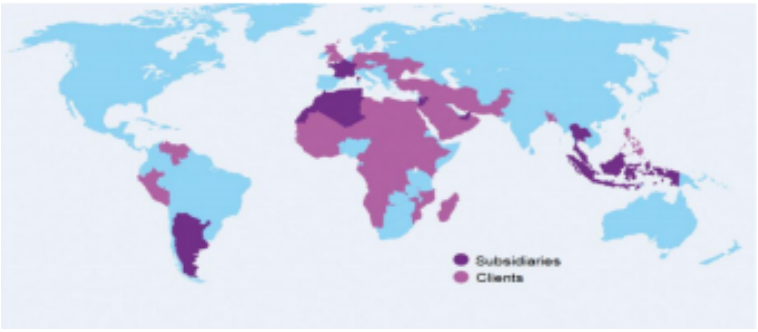

Figure 1.1: Geographical distribution of Sofrecom around the world.

1.1.2 Sector of activity

Sofrecom was able to confirm its experience in several sectors such as:

- Consulting
- Innovation
- Development
- Engineering
- Security
- Architecture
- Support and maintenance

1.2 Context of the project

This section is a general presentation of the scope of the project. It will start by its context , and the current situation, followed by problem statement and ending with the proposed solution.

1.2.1 General context of the project :

Sofrecom is conducting a project, called, VIO, which is a project that ensures the indirect sales to different telecom operators for example SFR, Bouygues Télécom, and Free . It is based on 4 java applications running in order to preform different roles, below is the detailed role of each :

- FOP: (Frontal Operator): responsible for taking into account orders issued by third-party mobiles operators (SFR, Bouygues Télécom, Free, etc.). It also tracks orders it has received by sending acknowledgments of receipt and delivery reports to the initial issuers.

- PARC : Repository of the commercial park of different wholesale offers such as: HD, FTTH, VGA, VGT.
- SAGIC: Acts as an exchange system for the order and delivery process. 50,000 orders are processed per day. SAGIC receives orders from many front-ends (including FOP), then it checks them, analyzes them and, according to criteria, routes them to the delivery application likely to handle them. SAGIC also manages the distribution of delivery reports issued by delivery applications.
- VIGIE: Infocenter of the VIO domain, where we can find all the historical data related to the VIO Project.

The project is running on three different environment :
Qualification, preproduction and production . The qualification environment itself is composed of three environments :

- Q1 : Internal qualification
- Q2 : Functional tests
- Q3 : Development

Figure 1.2 describes the architecture of the Qualification 1 environment.

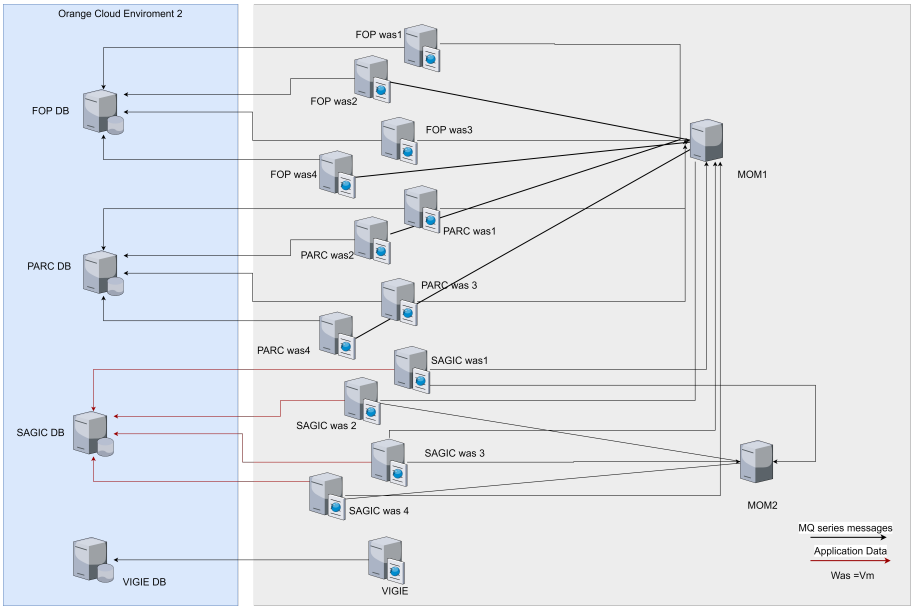

Figure 1.2: Qualification environment

- The same instance of FOP is running in 4 virtual machines, that is, to ensure load sharing, respectively, for PARC and SAGIC.
- The MOM(Message-Oriented Middleware) allows exchanging data and communication.

As for the Qualification 2 and 3 environment, they have nearly the same architecture as the Qualification 1 environment.

1.2.2 Problematic

In order to clearly see the problematic, let's see what any team member would encounter on a daily basis.
Starting his day, the developer discovers that the application is down and he can't figure out the cause behind the problem. He tries to open logs files, keeps scrolling searching for a hint ,He runs several commands in order to know the status of different services, through different VMs. Thus he wastes his time and cannot even report the real cause to the project manager. Causing the company a big loss of time and money.
The major cause of a similar situation is the lack of a monitoring and alerting system that can ensure that every time a problem occurs whether in the infrastructure or in the applications ,team members will be notified and have full and continuous visibility .

1.2.3 Proposed Solution

The goal of this project is to build a centralized set of dashboards where we can correlate metrics and log files. These dashboards must offer visibility across the entire infrastructure and keep the team members and leaders updated about the state of the running applications. The monitoring solution must provide continuous alerting based on different levels of severity .
It is also important to ensure deployment is prepared across the three environments (qualification, pre-production and production) ,through the automation of the installation and the configuration of each component of the monitoring stack .

1.3 Monitoring

Monitoring is a crucial step for any operating system or application; it brings many benefits. In fact, it helps identify problems early and thus prevent disasters and reduce downtime that can lead to business loss.

Monitoring can be classified into different categories based on the way data is collected and processed. It system-based, application-based or network-based.

1.3.1 Direct / Indirect monitoring classification

Monitoring can be classified into two categories, direct and indirect monitoring.

1.3.1.1 Direct monitoring

Monitoring can be direct whenever we pull data directly from the system, which means that there is approximately no delay between the pulled and visualized data.

Direct Monitoring is crucial to infrastructure health. It helps identify the exact time when an incident occurs and addressing it immediately.

1.3.1.2 Indirect monitoring

Indirect monitoring is when we base our monitoring on additional sources other than the system itself, which is the case of using log files as a source of supervision and alerts generation. In fact, we use indirect monitoring when it is impossible to instrument a certain application or service. As a consequence, we resort to logs to detect incidents. Every action that occurs in your system is recorded. Based on these log files, indirect monitoring provides the information you need to spot failures within your different applications or services, better understand already occured incidents, and identify different anomalies related to performance, compliance, or security.
This approach fails to detect certain incidents in real time. However, the effect of this delay would depend on the nature of the failure.

1.3.2 Agent/Agentless monitoring classification

Choosing the way you want to collect your data is a crucial step in every monitoring solution. You can use agents as intermediaries or opt for an agent-less model [2].

1.3.2.1 Agent-based monitoring

Agent-based monitoring uses a software component, usually installed on a client server, to collect data. The data are then returned to the monitoring server according to policies within the local agent, or upon requests from the monitoring instance. As a best practice, the agent responds based on different requests from the monitoring instance. Thus, the agent becomes very lightweight and has access to detailed metrics for better monitoring and reporting, which helps to get a deeper root cause analysis and better troubleshooting.

1.3.2.2 Agentless monitoring

Agentless monitoring is used to monitor various IT components. In fact, without having to install or manage an agent, various parts of the infrastructure can be supervised. Protocols or APIs are used to collect the same metrics that agents would collect in an agent-based model.

1.3.3 Datasource-based Classification

Monitoring can be classified based on datasources, which can be system, network, or application related.

1.3.3.1 System monitoring

It is the continuous monitoring of an IT infrastructure that involves gathering and collecting data related to different systems.
System monitoring includes monitoring resources such as: //Memory ,Disk ,services and processes.

1.3.3.2 Network monitoring

A network monitoring system includes software and hardware tools that can track various aspects of the network and its operation, such as:

- Traffic exchange.
- Bandwidth usage, and uptime.

These systems can detect devices and other elements that cross or touch the network and provide status updates. Network administrators rely on network monitoring systems to help them quickly identify problems such as equipment or link failures, or traffic congestion that limits data flow. The ability to detect problems extends to parts of the network that traditionally extend beyond its boundaries.

1.3.3.3 Web application monitoring

Web application monitoring is making sure that applications are running as they are supposed to. However, a web application can be available and still generate different errors. Thus, the process involves monitoring different aspects of the application :

- Application availability.
- Application dependencies (Databases).
- Application errors.

1.3.4 Protocols of monitoring

Several protocols are used to ensure systems supervision and monitoring.The most used ones are :

- **SNMP** is a standard protocol that queries related objects to extract data from switches, wireless controllers, servers, printers, routers, modems, etc. The collected data are used to develop information for monitoring system and network performance based on interface status, CPU usage, bandwidth usage, network latency, etc.[3]
- **ICMP** is a network protocol that allows error reporting. Network devices such as routers use ICMP to send error messages in situations such as when the host or client is unreachable or the requested information is unavailable.[4]

- **HTTP** is used to check websites' availability and ensure that the contents of the address match a specific criteria.

1.3.5 Monitoring and log management solution

Every monitoring solution must be composed of the essential elements presented in the figure 1.3:

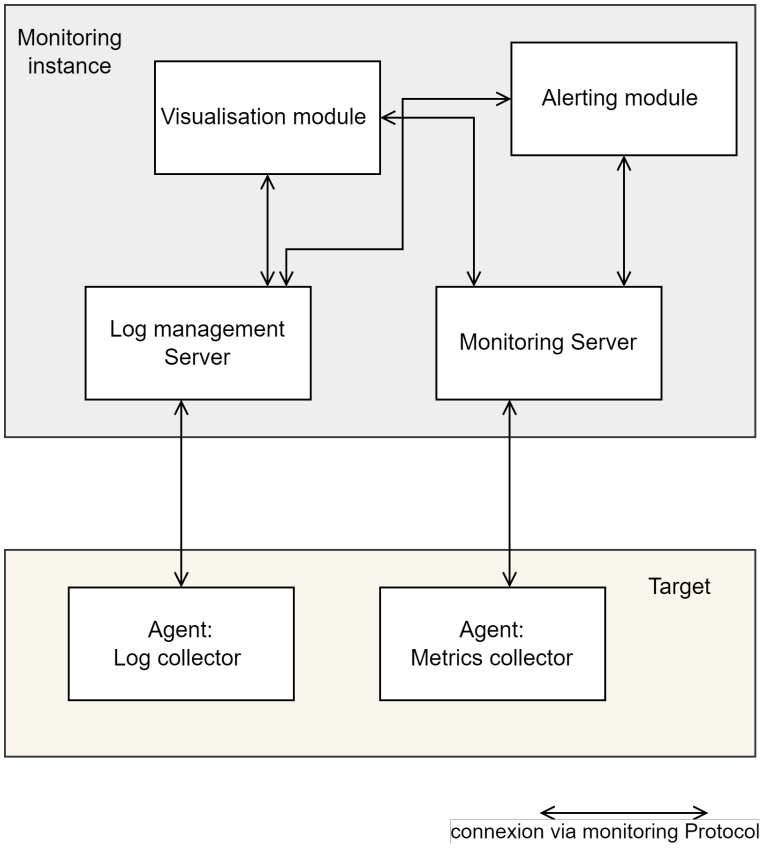

Figure 1.3: General architecture of monitoring solution

- **Log collector:** Agent responsible for log collection.
- **Metric collector:** Agent responsible for collecting metrics(from: system,network,application).
- **Monitoring server:** Responsible for managing monitoring metrics.
- **Log management server:** Stores and manages log streams.
- **Visualisation module:** Responsible for querying metrics and displaying through visuals(graphs, charts).
- **Alerting module:** Responsible for managing alerts and sending them to SMTP server.

1.3.6 Best practices of monitoring

To implement a strong monitoring system, first you have to define what resources you need to monitor and how you will monitor each one. You need to identify metrics that help you supervise the health of your infrastructure.
Finally, to keep your team informed and prevent possible incidents, you need to implement alerts throughout your system.

1.3.7 Metrics

Metrics represent raw data on resource usage or behavior that can be observed and collected within your infrastructure. As your infrastructure evolves, the type of information you need to monitor changes. In fact, there are several types of metrics you might need to track.

1.3.7.1 System metrics

System metrics can be low-level usage summaries provided by the operating system, or high-level data types related to a specific function or process. These are some of the important metrics for every server:

- CPU usage : Displays the total CPU usage, the user/system CPU usage h
- Memory: Displays the total amount of system memory currently used, cached, buffered, or free.
- Disk usage and available space within your machines.
- Services/processes : Displays the running/failing services and processes within your machines.

1.3.7.2 Applications metrics

In order to measure the performance of your operating application, several metrics must be monitored: Here are some of the important ones:

- **Uptime:** Monitors the availability of your app through different endpoints.
- **Error rates/Exception rates:** Measures how often does your application generate errors.
- **DNS lookup:** Measures DNS lookup time while the browser is loading the page.
- **Request rate:** Measures the traffic your application receives.

1.3.7.3 Network performance metrics

In order to optimize network performance, there are several metrics that need to be monitored. For example:

- **Bandwidth usage :** Measures the amount of data transmitted over a network packet loss: determining the number of packets sent successfully but never made it to the desired destination.
- **Packet loss :** Measures the number of packets sent successfully but never made it to the desired destination.
- **Availability:** Monitors whether or not a network is available and currently operational.

1.3.7.4 Server pool metrics

Instead of looking for metrics for each server or virtual machine on its own, it's better to see the global image of an infrastructure and monitor server pool metrics like monitoring the total number of instances and tracking which ones are down. The main purpose of an alert is to draw attention to the current state of the system. In fact, alerting is the part where the importance of monitoring solution is evaluated to make sure if the chosen metrics helped to enhance the system performance.
Alerting is the responsive component in which there is an action made based on metrics state or value, besides alerting helps disengage the administrators from the physical system, everything is centralized and summarized in notification alerts

1.3.8 Best practices and types of alerting

Few actions must be implemented within alerting strategy in order to have a powerful monitoring stack [5]:

- Communicate with the system administrators in order to choose the right threshold for every alert to avoid unsolicited alerting.
- Include important information in notification alerts so the receiver know what triggered the alert and where it came from
- To avoid spamming the receivers, fine-tune the alert rate and modify the frequency of the alerts.
- Consider muting alerts or rules when there is a planned downtime or maintenance.

Types

Types of alerts would differs based on the severity of the event. There is:

- A critical alert that demands immediate intervention or investigation from the support team and has no delay tolerance .It is considered a reactive alert.
- A warning alert sent when an issue occurred and it may not be causing a problem yet, it's the type of alert to avoid future problem.
- An information alert is an informative alert where there is no action required.

1.3.9 Comparative analysis of monitoring tools

In our case, as presented previously, the VIO project is based on hundreds of virtual machines. It can be a difficult and time-consuming task to keep up with such a huge number of VMs. Besides, it's essential to know the state of every single VM. For example, if a certain machine is consuming more CPU than it's supposed to be or if it's running out of disk space, the system administrator must be notified. That's why a monitoring tool is crucial to be deployed for the VIO project.

Now we are going to discover different tools used for monitoring and querying metrics, then we go through a comparative analysis in order to choose the solution that suits the needs of our project .
There are several open source monitoring tools. Here are the most performant and most used solutions:

- **Zabbix:** It is a monitoring solution for IT services and components, such as servers, networks, and virtual machines. It is suitable for any small or large.
- **Graphite:** It is an open source monitoring tool that stores numeric time series data and renders graphs based on this data [6].
- **Prometheus:** It is an open source monitoring solution that stores organizes, and queries metrics based on time series data.

To choose the most suitable tool, we will compare these tools based on how they process data from collection to querying.

Data collection

- Prometheus automatically discovers its targets and once configured successfully, data will be pulled periodically by the monitoring instance from the system; In the case that the targets do not expose metrics for an extended period of time, data can be pushed to Prometheus.
- Zabbix is based on server-agent architecture where the agents push data to the server; an agentless configuration can also be used to pull data instead of pushing it [7].
- Graphite has a special agent that passively listens to the data to be received, and for data collection additional solutions such as Fluentd or Statd must be included [8].

Data storage

- Prometheus has its own Time Series Database (TSDB) It can store and process more metrics than any other monitoring tool. Prometheus stores data internally or through an external DB for long-term storage. It provides key-value pairs, called labels, attached to a metric name, providing powerful organization and query capabilities. In addition, we can apply different types of retention to internally stored data, based on size, period, or both.
- Graphite uses its own TSDB like Prometheus; however, it stores its data with dot-separated model compared to key value tags used in prometheus.
- Zabbix [9] doesn't store data on its own; it requires an additional database (MYSQL PostgreSQL ..).

Data querying

Zabbix uses a method based on item keys, so we need to use SQL to fetch data from MySQL or PostgreSQL. As for Prometheus, it provides its own language, called PromQl, which allows the user to select time-series data in real-time. Finally, Graphite does not have a specific language; however, it applies functions to one metric at a time.
Table 1.1 summarizes the comparison between the different tools:

	Prometheus	Graphite	Zabbix
OpenSource	✓	✓	✓
Data model/storage	-Internal:key-value -Remote storage	-Internal:dot-separated -Remote storage	-Remote storage
Push /Pull approach	Pull	Push	Push
Query language	PromQL	Functions	Item -keys

Table 1.1: Comparison of monitoring tools

Choosing monitoring tool

To collect data in our case, in which we have a huge number of virtual machines to monitor, a pull-based approach would be the best. In fact, pulling data requires an extra configuration on the monitoring instance, which is way better than configuring agents on every single VM. In addition, it's easier to detect if a target is down in a pull-based system. Indeed, the monitoring system would notice that a certain VM is unreachable, and thus we avoid a possible data loss. As for the storage, to monitor VIO project, we are going to use internal

storage where we can have the ability to delete old data or to send it to archives for eventual use [10] and [11].

Prometheus architecture

Prometheus architecture consists of multiple components [12]. The most important ones are:

- **Prometheus server:** pulls/scrapes and stores time series data.
- **Special-purpose exporters:** Collect data from different nodes/targets, the most used ones are node exporter, which collects and exposes hardware ad kernel-based metrics, and black box exporter which allows users to monitor HTTP, DNS, TCP and ICMP endpoints.
- **Alertmanager:** handles alerting.
- **Push gateway:** receives data from short-lived jobs.
- **Client libraries:** instrument application code.

The figure 1.4 below shows the architecture of Prometheus components.

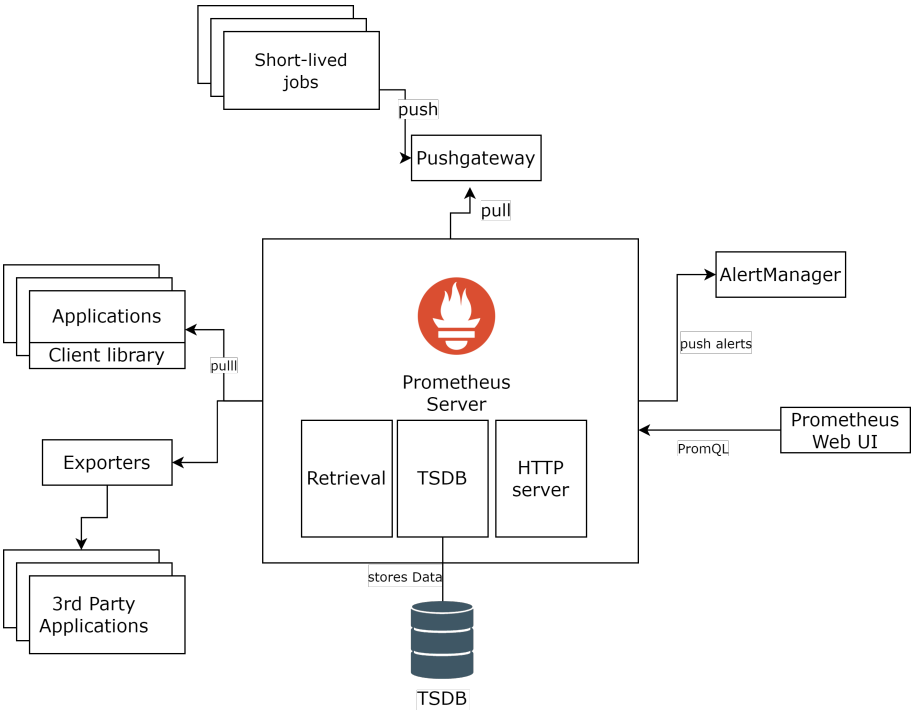

Figure 1.4: Architecture of Prometheus

1.3.9.1 Comparative analysis of log centralization and analysis tools

To ensure optimal performance of the project VIO, adding a log management tool to our monitoring stack along with Prometheus would be the best option.
We are going to base our choice on the following criteria:

- The solution must be opensource. It should be compatible with Prometheus, it must use resources sparingly and finally it works with different types of log files and formats.

Graylog

Is an open-source used for collecting, indexing, and managing log files. It has a complex architecture based on 3 basic components: the Graylog server for visualization and metric querying, MongoDB to manage metadata, and finally ElasticSearch for storing logs [13].

ElasticSearch

Elasticsearch is an open-source tool for log analysis and management. It offers a powerful platform for collecting and processing data from different data sources. However, it may need more resources than the usual log management tools [14].

Loki

is an open-source log aggregation tool, designed to be cost-effective and use fewer resources as it does not index the whole log content. However, it uses labels for every log stream. In addition, it uses a data model and a query language very similar to the ones used by Prometheus. The table below summarizes the comparison between the different tools:

	Graylog	ElasticSearch	Loki
OpenSource	✓	✓	✓
Installation/ configuration	Simple	Simple	Simple
Ressource Usage	Moderate	High (50%)	Moderate
Compatibility with Prometheus	No	No	Yes (very similar query language)

Table 1.2: Comparison of log managment tools

Choice

To conclude ,obviously Loki is the one that suits the needs of our project the best ,it's open source , it fits perfectly with Prometheus besides ,having moderate resource usage will go long with the fact of using internal storage .

Loki architecture

Loki architecture is based on different components each one of them plays a specific role :

- Distributor : handles incoming streams in a the writing path.
- Ingester : creates different filters and indexes.
- Querier : responsible for looking up the requested trace ID in internal or backend storage.
- Query Frontend : shards the search space for an incoming query. Traces are exposed via an HTTP endpoint: GET /api/traces/<traceID>

Figure 1.5 below shows the architecture of Loki components and the read/write data flow [15].

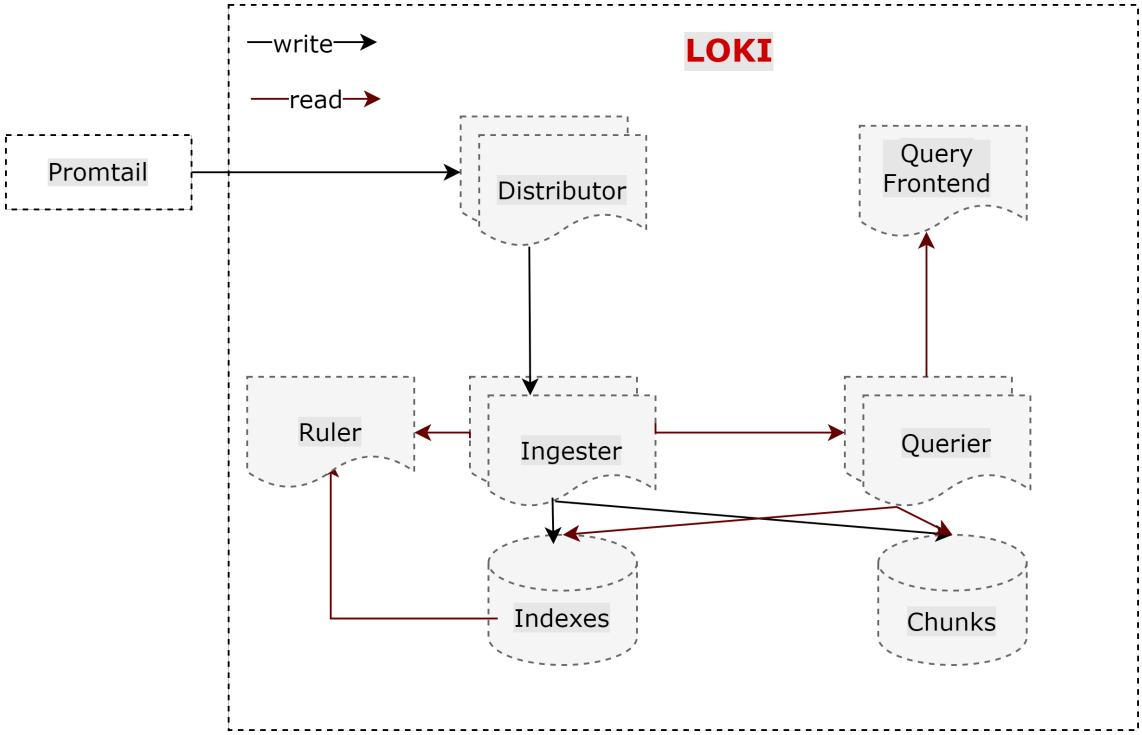

Figure 1.5: Architecture of Loki

1.3.9.2 Comparative analysis of visualisation tools

kibana

Is a part of the ELK stack. It offers a tool to explore the data, it allows users to create powerful dashboards using different ways of graphs, graphs, tables, or even geographical maps. It also allows users to search for a specific event through its indexed data. It is an open source tool, but certain functionalities require purchasing a license [16].

Grafana

Is an open source visualization tool that connects to a variety of data sources. It also offers different types of visualization, such as graphs, charts, heat maps, and histograms. It is a really powerful and user-friendly tool for time series analysis. It allows easy editing, in fact, it does not require a lot of skills to learn how to manipulate different dashborads.[17]

Choosing the visualization tool

Comparing the two options, Grafana seems to be the most suitable visualization tool, because it is compatible with both, Prometheus and Loki and offers many functionalities that enable us to create captivating visuals.

1.4 Automation

Automation is the art of making a process more effective and efficient. It is based on a set of logic commands combined with equipment that executes the commands to achieve a set of instructions. Automation is required whenever we need to deploy the same instructions through different machines. For example, it makes the process of installing the same agent across the entire infrastructure easy and simple.

Ansible

Ansible is simply an open source computer engine that automates the deployment of applications and services.
Ansible is easy to deploy because it does not use personalized security agents or infrastructures[18].
Ansible uses a playbook to describe automation tasks, and the playbook itself uses a simple language, namely Yaml, which is easy to understand. As a result, even the heads of IT infrastructure support can read and understand the playbook, as well as debug it if necessary.
Ansible doesn't use an agent, which means that it uses SSH to connect by default to different nodes.
After connecting to the different nodes, Ansible pushes small programs called "Ansible modules" and executes these modules.

ODE

The Orange Deployment Engine is a secure deployment solution for Ansible playbooks. It's mainly used to run Ansible playbooks on non-PROD and PROD servers, without requesting stream openings. ODE is natively compatible with Orange France's cloud and on-premises servers.
The ODE fits into continuous deployment chains under Gitlab-CI, which is a continuous integration/continuous deployment tool used for software development.

How does ODE works ?

After a job or pipeline has been triggered from Gitlab-CI, ODE will perform the following actions:

- Generate a unique id for the deployment.
- Check the user's permissions for the application.
- Start a base dedicated to the deployment.
- Transfer the files necessary for the deployment (inventory, playbook, war, etc.) on the base.
- Run the Ansible playbook with the provided parameters.
- Show ODE and Ansible logs in Gitlab-CI.
- Stop the base at the end of the deployment.

Automation will be applied through this project in order to facilitate the configuration and installation of new components.

1.5 Conclusion

In this chapter, we introduced the host company and discovered the different concepts to be implemented through the project. We also presented a comparative analysis of different tools to finally choose the ones to use for this solution. The next chapter will introduce the overall architecture and design of the entire project.

Chapter 2

Requirements analysis and solution presentation

Introduction

Any functional system is realized for the purpose of satisfying one or more requirements. This chapter defines the requirements and functionalities of the solution to be put in place, followed by an UML design of the project.

2.1 Requirements analysis

In this part, we will identify the actors of our system and determine the requirements associated with the proposed solution.

2.1.1 Functional requirements

The functional requirements correspond to the functionalities of the system. These are the requirements that specify, in an informal way, the input and output behavior of the system. The objective of our project is to set up a centralized platform for the collection and analysis of metrics and logs. Therefore, our system must meet the following requirements:

- **Collection and querying of operating system metrics:**
  The system must collect different metrics of the operating system.
- **Collection, centralization and storage of logs:** The system must collect logs from different virtual machines in order to provide the necessary data for further processing, then centralize and store these logs on the monitoring instance.
- **Ensure application availability supervision:**
  Monitor different application endpoints continuously

- **Detect errors and warnings:**
  Extract the significant error messages and different java exceptions generated in applications logs in order to make the diagnostics much more easier by eliminating the non significant errors.
- **Log and metrics exploitation:**
  The system must :
  - be configured so that the refresh rate of the displayed data can be changed.
  - be able to allow the user to search the list of logs.
  - Allow the possibility of adding specific operating system metrics through cron-jobs.
  - Allow the possibility of searching for a specific pattern in logs.
  - Offer the possibility to all users to generate curves and graphs describing the state of the network and to make statistics according to their needs.
- **Installation and configuration automation**
  The solution must offer the possibility to add and configure new target automatically through automation engine, such as, Ansible.

2.1.2 Non-Functional requirements

By non-functional requirements we designate requirements for the smooth running of functional-requirements. Indeed, this type of requirements represents the various constraints that the system must check in order to integrate it into its environment.
In our case, the system must guarantee the following constraints :

- **Extensibility**
  The proposed architecture must allow the addition of future functionalities easily.
- **Ergonomics**
  The monitoring interface should be user-friendly, interactive, ergonomically consistent and well-adapted to use.
- **Rapidity**
  To ensure a gain in response time (application loading, screen opening, and refresh times) and processing time, the application must be fast during data loading and processing (functions, data exports).
- **Flexibility**
  Adding a new equipment should not be a difficult task. So the system must be able to accept new components easily.
- **Use of open-source tools**
  To set up the monitoring platform, it is necessary to use open source tools so that there is a large community to consult in case of troubleshooting.

2.2 Proposed solution

Our solution offers monitoring and supervision of VIO project.
This monitoring solution is entirely based on the use of agents and it satisfies both classification direct and indirect monitoring. In fact, system and application monitoring were implemented directly, the solution also includes log management, which falls under the indirect category.
We describe in the following each part of our solution.

Monitoring instance

We started by preparing the monitoring stack and having it ready to pull/receive data, as we can see in Figure 2.1 below, which describes how the monitoring instance communicates with the virtual machines running in the qualification environment.

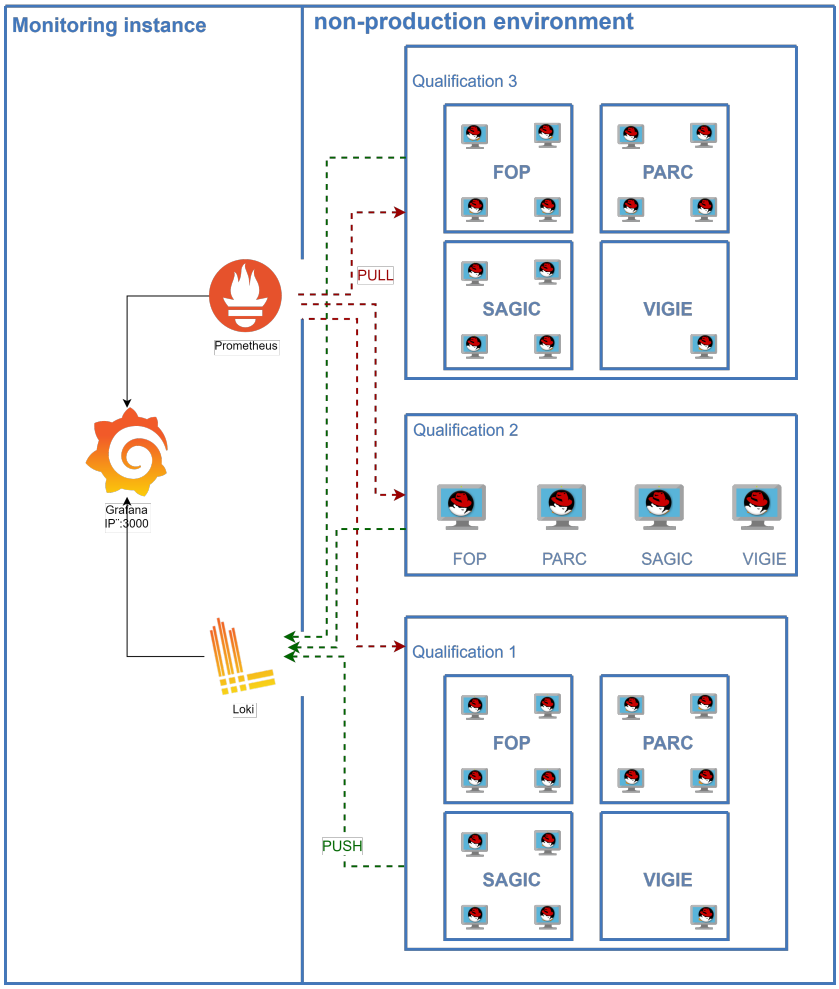

Figure 2.1: Monitoring instance communication with qualification environment

We imclude different aspects of monitoring :

System monitoring

We collect system metrics from different VMs deployed on Orange Fast Cloud on which the VIO project's applications are running. We expect the deployment and configuration of a node exporter on every VM so that system-related metrics can be exposed for Prometheus to be pulled and then be sent to the Grafana instance. We also provide cron jobs that periodically send system-related metrics that the node exporter does not provide.

Log monitoring

The second step is to collect and centralize application logs from different VMs. In our case, the same log file can contain logs about databases or queues or simply application logs. These log files are periodically centralized in the corresponding VM application. Our solution is to centralize these files at the monitoring instance so we can later process them and query relevant metrics.

Application monitoring

To monitor each application, our solution is to consider each application as a blackbox and focus on the application availability. We configure a blackbox exporter on each virtual machine .In fact, a blackbox exporter is an agent responsible for exposing metrics about application endpoints status, it allows blackbox probing of endpoints over HTTP, HTTPS, DNS, and other network protocols.Thus we ensure that the application is working as it should. Figure 2.2 gives a closer look at how our proposed solution is implemented on each virtual machine.

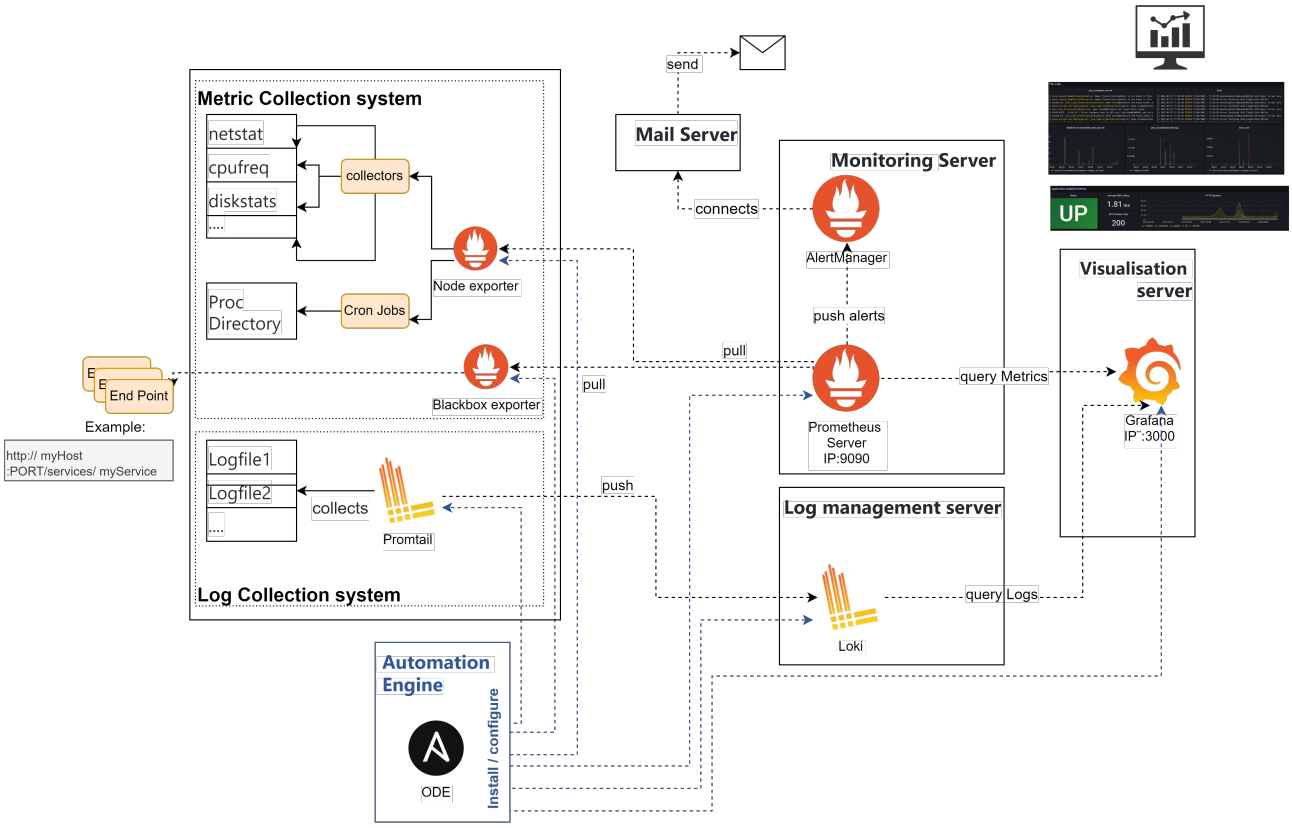

Figure 2.2: Monitoring instance communication with VM taget

After accomplishing all these tasks, the pulled metrics will be displayed in a dashboard using Grafana.

2.3 UML design

In this section, we will introduce the design of our application.

2.3.1 Actors identification

We identify three actors in our project:

- **Team member:** This actor has the ability to create dashboards, add specific metrics, and exploit log files.
- **Administrator:** This actor is actually a team member but with more privileges. He is the one responsible for managing different users and running all Ansible playbooks through a triggered pipeline.
- **Project manager:** This actor is in charge of following up or consulting the various dashboards.

2.3.2 General Use Case diagram

Use case diagrams model what the system is used for, by organizing the possible interactions with the actors. For this, we describe the main functionalities of our project by specifying the actions that each actor in our system can perform, either through the dashboard or through the deployment engine.

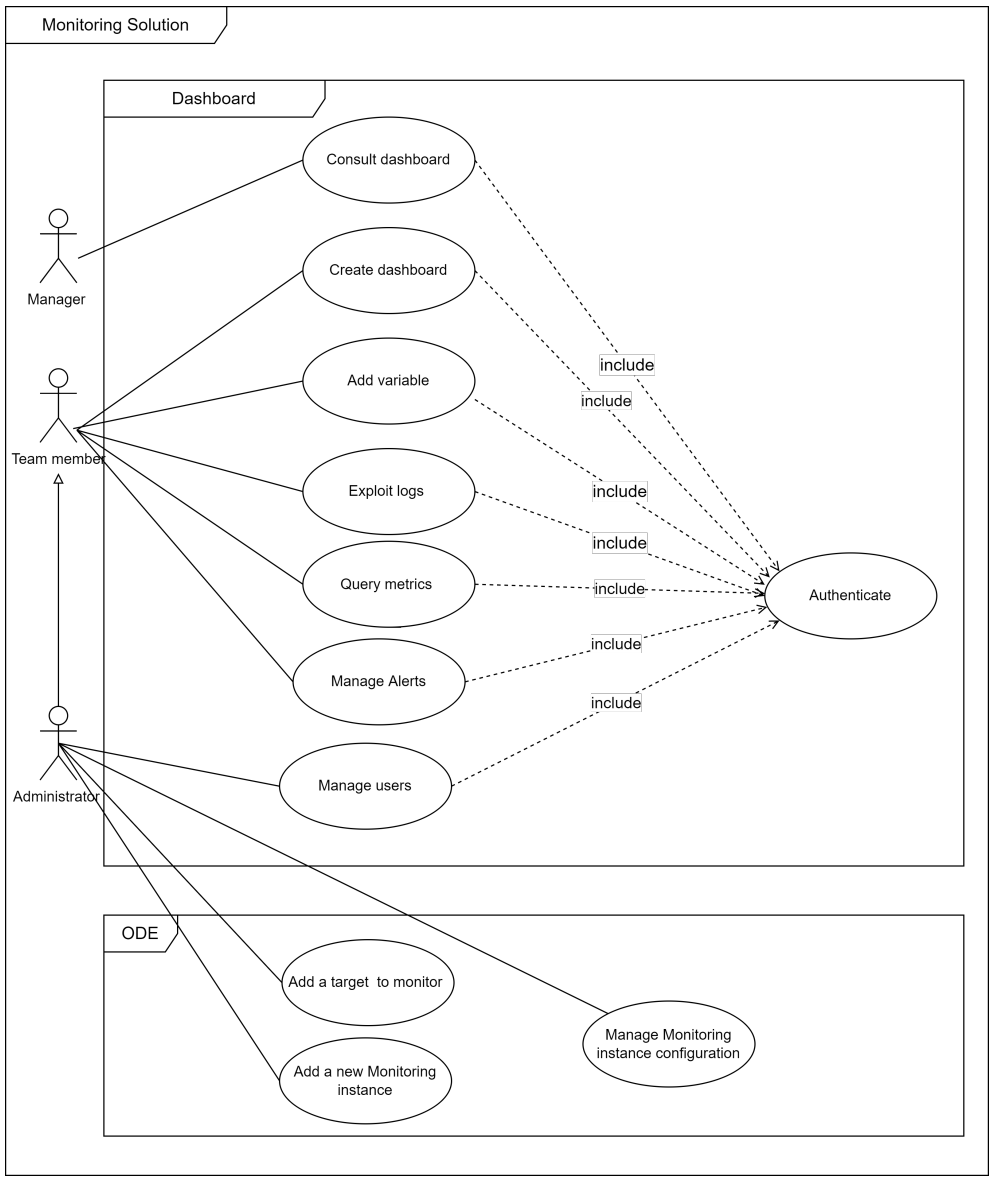

Figure 2.3: General Use case diagram

Our solution is used by three actors. Authentication is required for each actor. First, the manager he can only consult the different dashboards.Second, team members can either create dashboard, add variable to make the dashboard more dynamic, exploit log files, query metrics, and manage alerts.Finally, the administrator whose actually a team member can also manage users, add or configure a new target through the automation engine.

2.3.3 Class diagram

A class diagram is the best method to understand each part of our solution and what functions can each component execute.

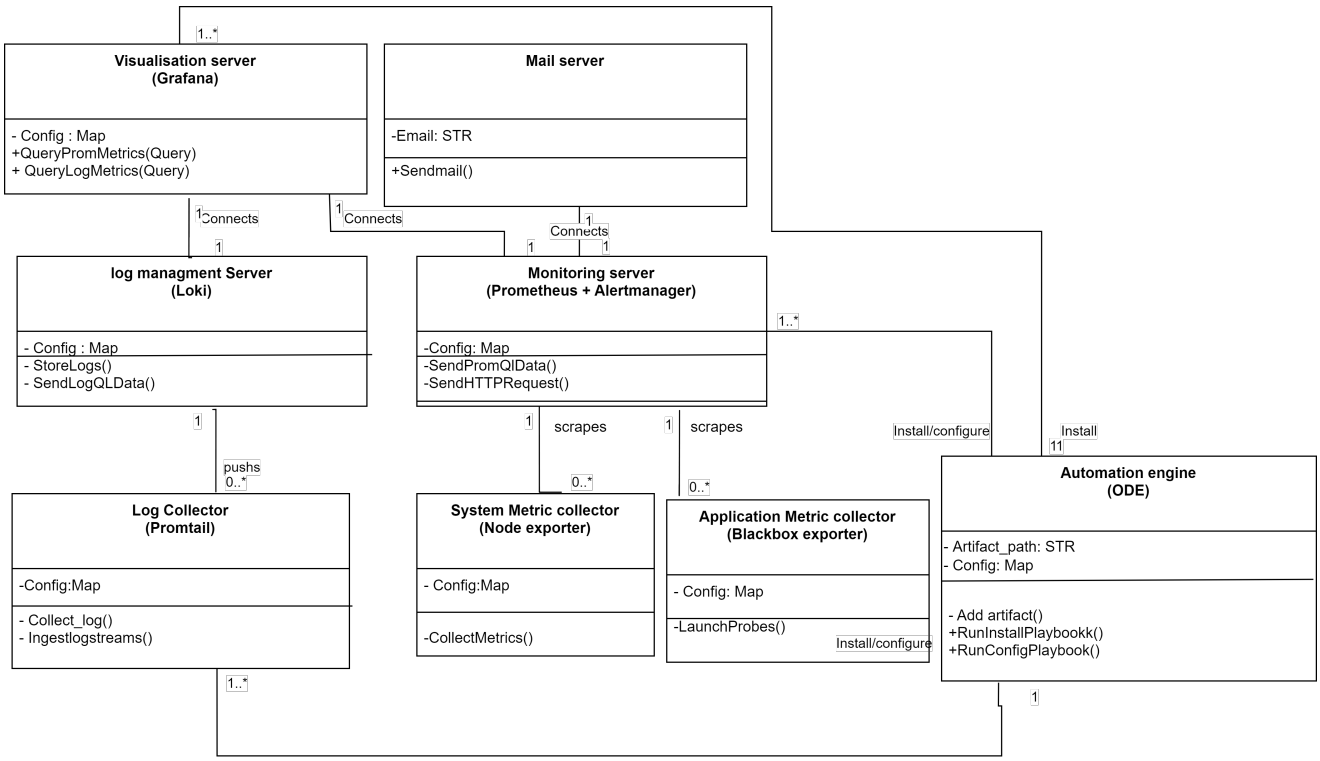

Figure 2.4: Class diagram

The class diagram is composed of:

- Visualisation server: it has two public functions QueryPromMetrics and QueryLogMetrics. Both functions take as a parameter the query to compute.It has a private attribute which is its configuration.
- Monitoring server: it has two private functions SendHTTPRequest and SendPromQlData. It also has a private attribute which is its configuration.
- Log management server it has two functions StoreLogs() and SendLogData(). It also has a private attribut which is its configuration.

- System-Metric Collector: it has a private function which is CollectMetrics(). It also has a private attribute which is its configuration.
- Application-Metric Collector: it has a private function which is LaunchProbes(). It also has a private attribute which is its configuration.
- Log Collector: it has two private functions which are CollectLog()and IngestLogStream(). It also has a private attribute which is its configuration.
- Automation Engine: it has two private functions which are RuninstallPlaybbok(), and RunConfigPlaybook(). Besides it has a public function Add-artifact(). It also has a private attribute which is its configuration and Artifact-path which is used to collect necessary artifacts.
- Mail Server: it has a public function which is Sendmail(). It also has a private attribute which is the Email address.

In fact, a single monitoring Server can scrape multiple collectors. Besides multiple log collectors can push to the log management server. From another side, in our proposed solution, a single visualisation server can connect to a single monitoring server and a single log management. The automation engine can install or configure multiple components of our solution. server

2.3.4 Activity diagram

The activity diagram is the behavioral diagram of UML, allowing us to represent the triggering of events based on system states. The following diagram represents the different stages that allow us to produce a dashboard.

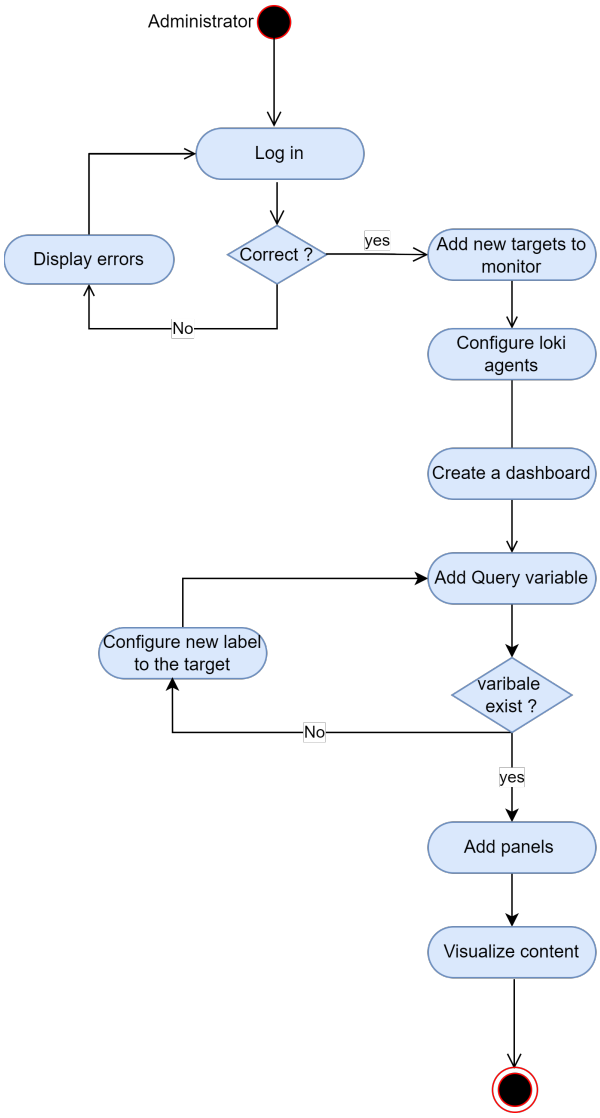

Figure 2.5: Activity diagram

To use our solution, the administrator must first successfully log in to the dashboard interface. Then, he begins by automatically adding new targets via the automation engine; once the virtual machines are successfully monitored, he can create variables and add panels. The final result is a dashboard to monitor various VMs.

2.3.5 Sequence diagrams

We provide in the following a series of sequence diagrams to put our class diagram into action and visualize different communications between different components.

2.3.5.1 Sequence diagram: Prometheus metric collection

This scenario 2.6 demonstrates how metrics are being collected by our monitoring server's pull-requests from different exporters.

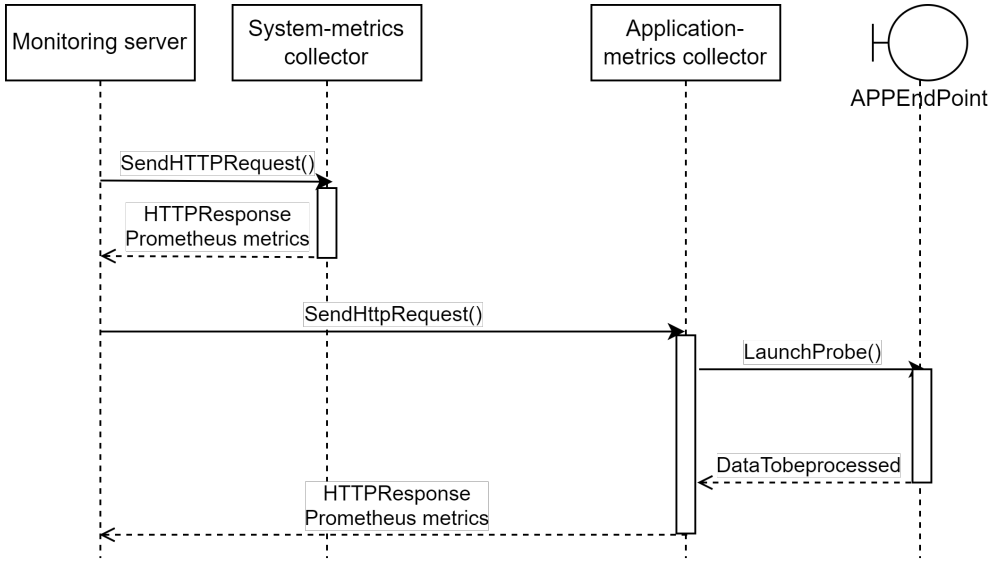

Figure 2.6: Prometheus metrics collection sequence diagram

In fact, the monitoring server sends http requests to different metrics collectors every 5s. The Application-metrics collector launches probes which are scraping requests to different Application endpoints and then responds to the monitoring server through http responses.

2.3.5.2 Sequence diagram:log collection

This scenario 2.7 explains how Promtail collects log files and pushes them to Loki:

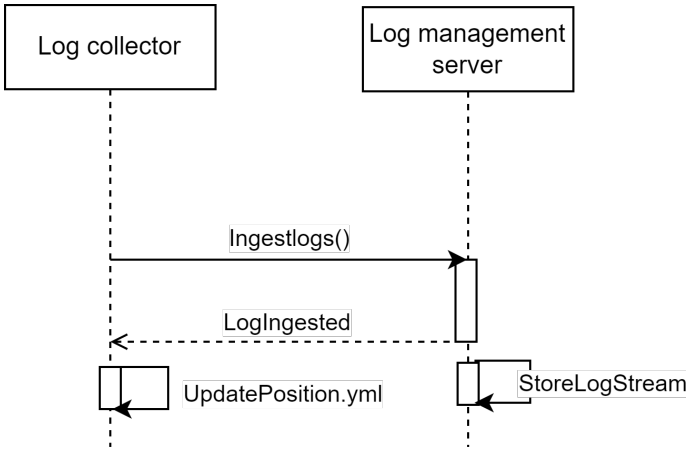

Figure 2.7: Log collection sequence diagram

Log collector, ingests log streams to the log management server where they are stored.

Once log stream is ingested successfully, the log collector updates its position.yml which is a file where the log collector keeps records about log ingestion.

2.3.5.3 Sequence diagram: Querying in Grafana

The scenario 2.8 below describes how metrics are being queried and visualized in Grafana :

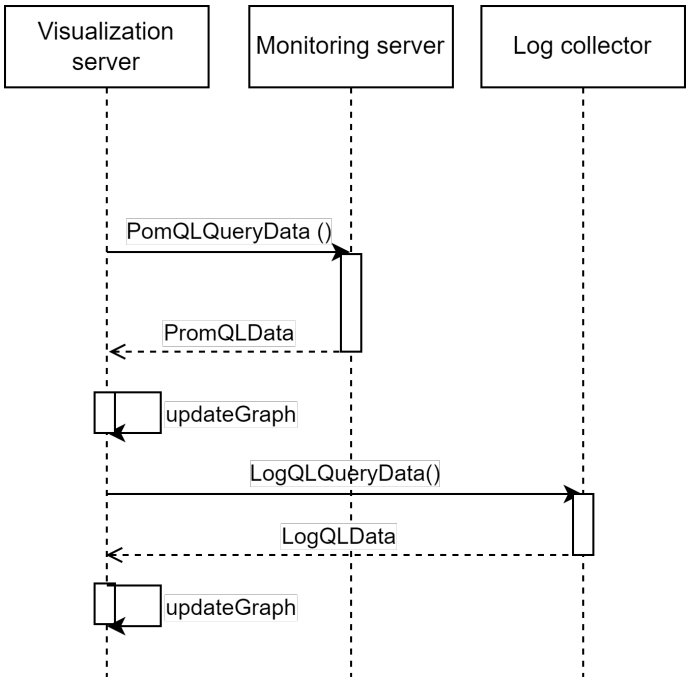

Figure 2.8: Querying in Grafana sequence diagram

Once we add a panel in our dashboard, we use PromQl/LogQL expressions, the visualization server sends query requests to monitoring and log management servers to compute the output and thus update the graphs.

2.3.5.4 Sequence diagram: Automation

In order, to better understand what happens when we want to monitor new virtual machines and how automation makes it an easier and simpler process, Figure 2.9 shows using a sequence diagram how the automation engine, ODE, interacts with the administrator through Gitlab-Ci.

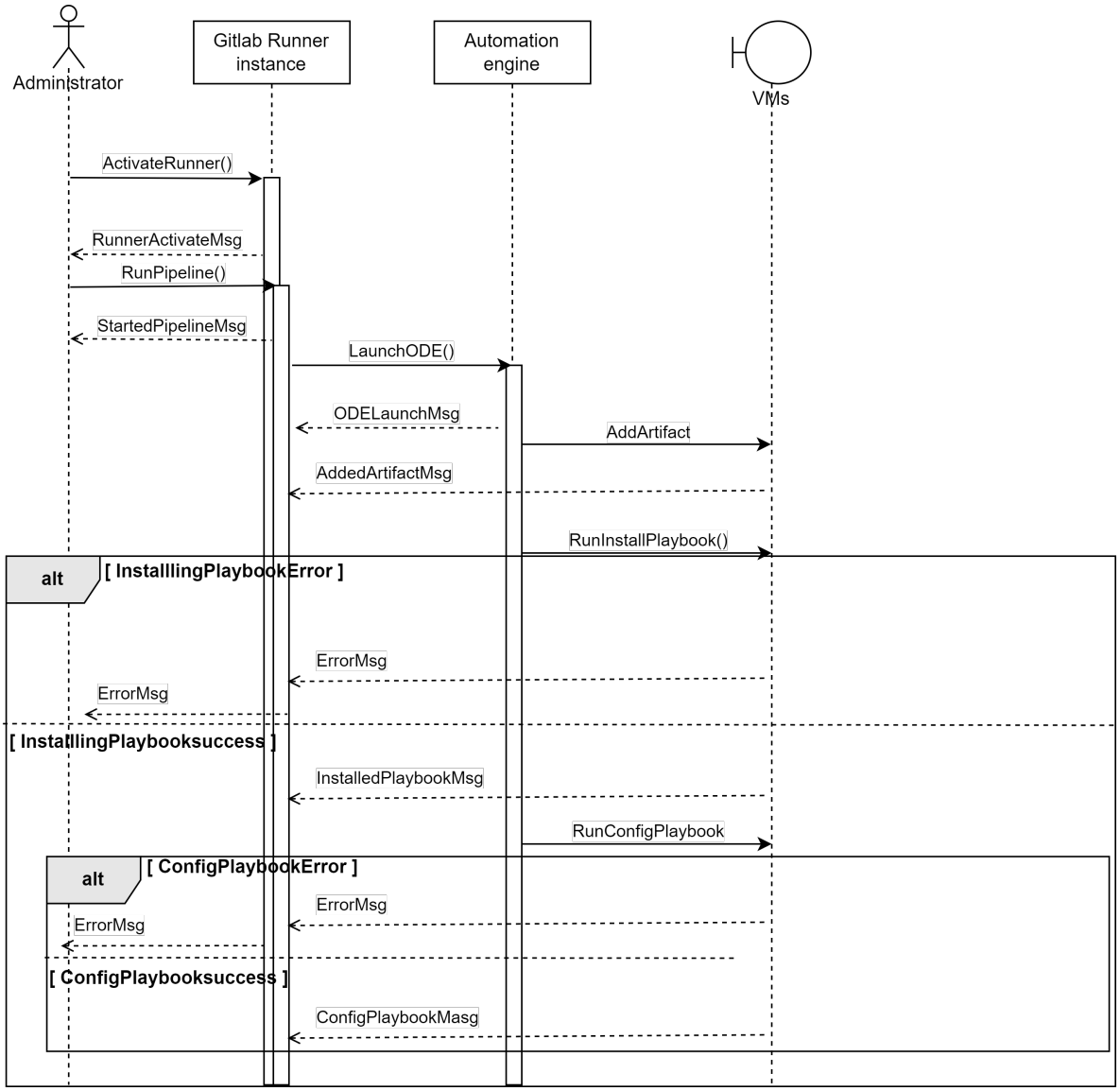

Figure 2.9: Automation sequence diagram

In order to automatically add a target, several actions are triggered. First, the administrator has to launch a pipeline through Gitlab-CI which activates a Gitlab Runner instance. This Runner is responsible for launching the automation engine, ODE.
ODE adds required artifacts on the virtual machines and runs the necessary playbooks. In case of problems during the process, error messages will appear on Gitlab-CI console.

2.3.6 Component diagram

The component diagram 2.10 below shows the system components and the type of connection between them.

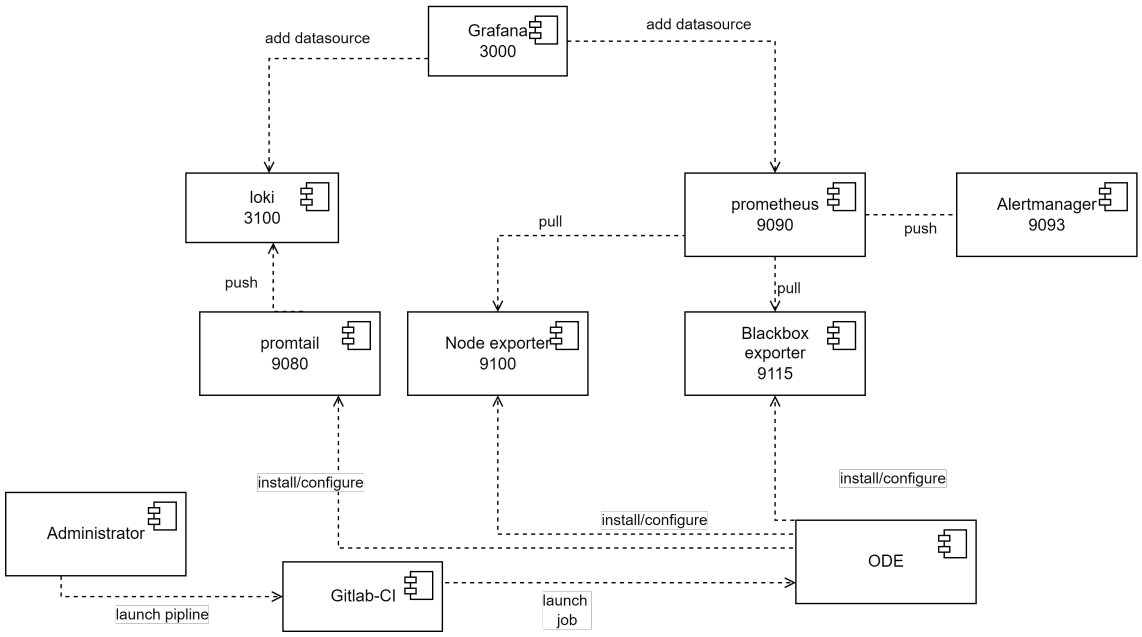

Figure 2.10: Component diagram

Our solution involves different components. In fact, Blackbox exporter and Node exporter, which are running on ports, 9115 and 9100 respectively, are responsible for exposing metrics for Prometheus server(running on port 9090). Promtail, running on port 9080, pushs log streams to log management server Loki (running on port 3100). Once we add Prometheus and Loki as datasources in Grafana, we have a running system, ready to create dashboards. From the other side, Gitlab-Ci is responsible for launching jobs to run ODE, our the automation engine. Then ODE takes the lead to install or configure different components.

2.3.7 Deployment diagram

The diagram 2.11 below shows the hardware and software components on which we deployed our solution and specifies which protocol is used in each connection.

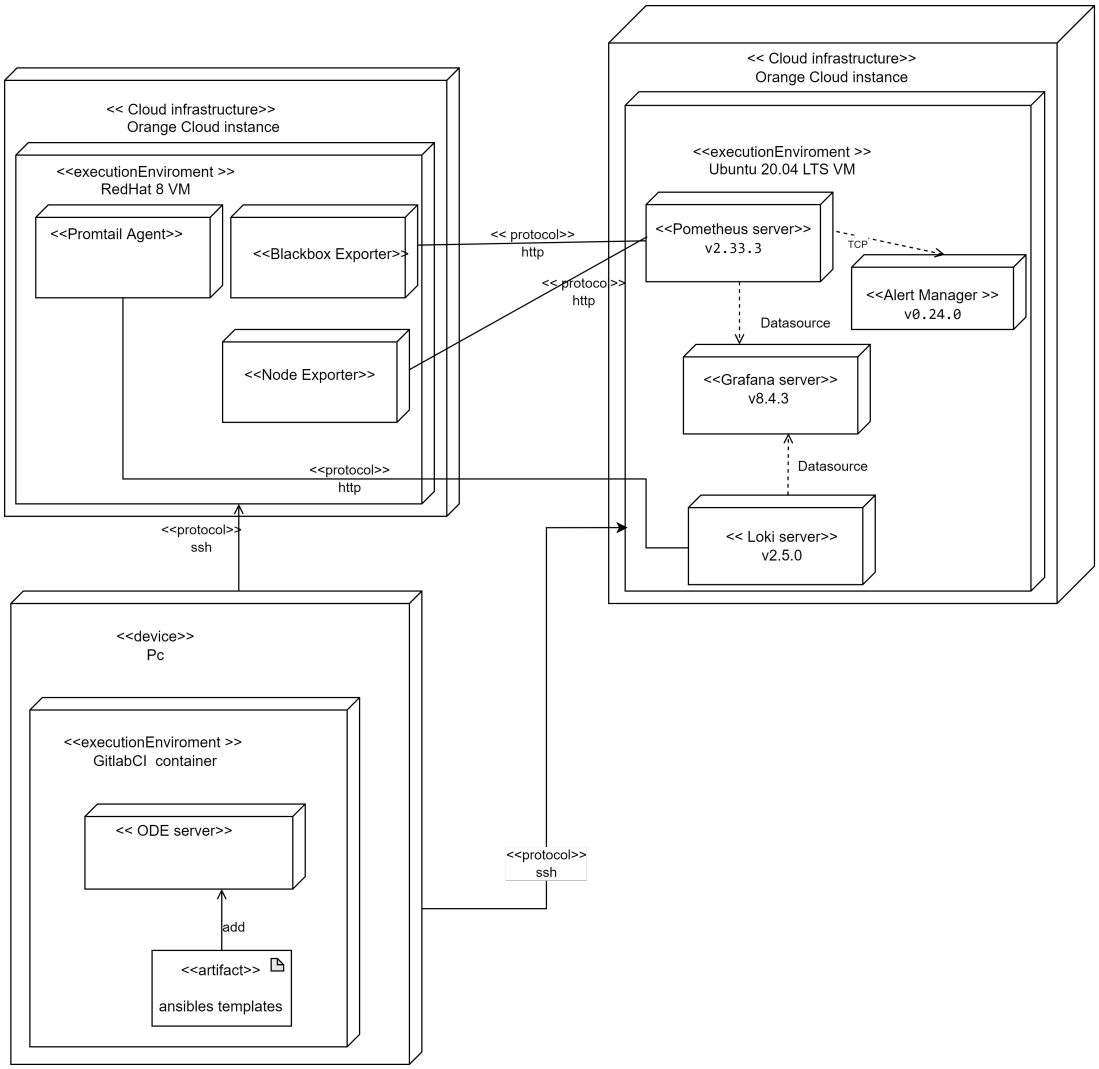

Figure 2.11: Deployment diagram

To deploy our solution we used three different types of environments.
sWe deployed the monitoring stack, which are Prometheus+Loki+Grafana+AlertManager, on an Orange Cloud instance with Ubuntu 20.04 as an operating system. As for the agents, Promtail and exporters, we deployed them on Vio's Orange Cloud instances. The automation engine is deployed on a Gitlab-CI container, which we access it through Giltab-CI pipline on our personal computer.

2.4 Conclusion

In this chapter, we explained the general architecture of our solution, followed by a detailed design through which we introduced different components of the system and the communication between them.

Chapter 3

Implementation

Introduction

In this chapter, we will present the achieved work. First, we will introduce the working environment. Then, we will present the implementation of our monitoring and log management solution by detailing its different parts.

3.1 Hardware/Software environment

Hardware

- The personal computer has the following configuration:
  - OS: Windows 10
  - PROCESSOR: Intel(R) Core(TM) i5-4310M 2.70 GHz
  - RAM: 16 GB
- **Monitoring VM**
  - OS: Ubuntu 20.04 LTS
  - vCPU: 2 vCPU
  - RAM: 3 GB RAM
- **Application VMs**
  - OS: Red Hat Enterprise Linux Server release 7.4
  - vCPU: 2 vCPU
  - RAM: 4 GB RAM

Software

MobaXterm

MobaXterm is a complete toolbox that provides access to a range of remote network tools. We used this tool to facilitate the access to different machines.

Figure 3.1: MobaXterm logo

Winscp

Winscp is an opensour SFTP/FTP client, we used this tool to transfer packages and files to our machines.

Figure 3.2: Winscp logo

visual Studio code

Visual Studio Code is a source-code editor. We used it to write configuration files, Ansible playbooks, and templates.

Figure 3.3: Visual Studio Code logo

3.2 Setup of the Work Environment

In this section we will present how we did set up each part of our system.

3.2.1 Setting up the monitoring instance

Setting up Prometheus

In order to have Prometheus running as it should, first, we need to create a configuration file under the name prometheus.yml. This file is to be modified each time we have a new target we want to monitor. In this file, we specify the following parameters:

- **Scrape interval:** How often we want the target to be scraped.

- **Scrape timeout:** How long until the scrape request times out.
- **Evaluation interval:** How long we evaluate the rules which are the basis of alerts.

```
root@dvviows00b0000p:/opt/prometheus# cat prometheus.yml
global:
    scrape_interval:     5s
    evaluation_interval: 5s
    scrape_timeout: 5s
alerting:
  alertmanagers:
  - static_configs:
    - targets: ['10.106.83.105:9093']
rule_files:
  - /opt/prometheus/rules.yml
scrape_configs:
- job_name: 'prometheus'
  scrape_interval: 10s
  static_configs:
    - targets: ['10.106.83.105:9090']
- job_name:
  scrape_interval:
  metrics_path:
  params:
```

Figure 3.4: Prometheus.yml

After adding the configuration file, we create a systemd unit file so we can easily manage Prometheus as a systemd service.
We specified retention options so we can manage the local storage of data collected by Prometheus.

```
root@dvviows00b0000p:/opt/prometheus# cat /etc/systemd/system/prometheus.service
[Unit]
Description=Monitoring system and time series database

[Service]
Restart=always
User=prometheus
ExecStart=/opt/prometheus/prometheus --config.file=/opt/prometheus/prometheus.yml  --storage.tsdb.path=/opt/prometheus/data --storage.
tsdb.retention.time=14d  --storage.tsdb.retention.size=5GB
ExecReload=/bin/kill -HUP $MAINPID
TimeoutStopSec=20s
SendSIGKILL=no
LimitNOFILE=8192

[Install]
WantedBy=multi-user.target
```

Figure 3.5: Prometheus systemd Unit file

Finally, Prometheus is running successfully, ready to collect metrics:

```
root@dvviows00b0000p:/opt/prometheus# systemctl status prometheus.service
● prometheus.service - Monitoring system and time series database
     Loaded: loaded (/etc/systemd/system/prometheus.service; enabled; vendor preset: enabled)
     Active: active (running) since Wed 2022-05-25 18:25:40 CEST; 1 weeks 3 days ago
   Main PID: 234089 (prometheus)
      Tasks: 9 (limit: 4576)
     Memory: 1.7G
     CGroup: /system.slice/prometheus.service
             └─234089 /opt/prometheus/prometheus --config.file=/opt/prometheus/prometheus.yml --storage.tsdb.pat

Jun 05 07:00:03 dvviows00b0000p prometheus[234089]: ts=2022-06-05T05:00:03.225Z caller=checkpoint.go:98 level=in
Jun 05 07:00:05 dvviows00b0000p prometheus[234089]: ts=2022-06-05T05:00:05.153Z caller=head.go:987 level=info co
Jun 05 07:00:07 dvviows00b0000p prometheus[234089]: ts=2022-06-05T05:00:07.605Z caller=compact.go:460 level=info
Jun 05 07:00:07 dvviows00b0000p prometheus[234089]: ts=2022-06-05T05:00:07.623Z caller=db.go:1287 level=info com
Jun 05 07:00:07 dvviows00b0000p prometheus[234089]: ts=2022-06-05T05:00:07.635Z caller=db.go:1287 level=info com
Jun 05 07:00:07 dvviows00b0000p prometheus[234089]: ts=2022-06-05T05:00:07.648Z caller=db.go:1287 level=info com
Jun 05 09:00:03 dvviows00b0000p prometheus[234089]: ts=2022-06-05T07:00:03.139Z caller=compact.go:519 level=info
Jun 05 09:00:03 dvviows00b0000p prometheus[234089]: ts=2022-06-05T07:00:03.193Z caller=head.go:818 level=info co
Jun 05 09:00:03 dvviows00b0000p prometheus[234089]: ts=2022-06-05T07:00:03.203Z caller=checkpoint.go:98 level=in
Jun 05 09:00:04 dvviows00b0000p prometheus[234089]: ts=2022-06-05T07:00:04.815Z caller=head.go:987 level=info co
lines 1-19/19 (END)
```

Figure 3.6: Running Prometheus

Setting up Loki

Same for Loki, we configured a systemd unit file and as we can see in the following figure that it's running successfully:

```
root@dvviows00b0000p:~# systemctl status loki.service
● loki.service - Loki log aggregation system
     Loaded: loaded (/etc/systemd/system/loki.service; enabled; vendor preset: enabled)
     Active: active (running) since Mon 2022-05-16 16:04:20 CEST; 2 weeks 6 days ago
   Main PID: 990 (loki)
      Tasks: 22 (limit: 4576)
     Memory: 113.1M
     CGroup: /system.slice/loki.service
             └─990 /opt/loki/loki --config.file=/opt/loki/loki-local-config.yaml

Jun 06 15:53:11 dvviows00b0000p loki[990]: level=info ts=2022-06-06T13:53:11.890545783Z caller=metrics.go:92 org_id=fake latency=fast >
Jun 06 15:53:11 dvviows00b0000p loki[990]: level=info ts=2022-06-06T13:53:11.89084873Z caller=metrics.go:92 org_id=fake latency=fast q>
Jun 06 15:53:11 dvviows00b0000p loki[990]: level=info ts=2022-06-06T13:53:11.891061603Z caller=metrics.go:92 org_id=fake latency=fast >
Jun 06 15:53:11 dvviows00b0000p loki[990]: level=info ts=2022-06-06T13:53:11.89145417Z caller=metrics.go:92 org_id=fake latency=fast q>
Jun 06 15:53:11 dvviows00b0000p loki[990]: level=info ts=2022-06-06T13:53:11.891779464Z caller=metrics.go:92 org_id=fake latency=fast >
Jun 06 15:53:11 dvviows00b0000p loki[990]: level=info ts=2022-06-06T13:53:11.893458884Z caller=metrics.go:92 org_id=fake latency=fast >
Jun 06 15:53:11 dvviows00b0000p loki[990]: level=info ts=2022-06-06T13:53:11.894592166Z caller=metrics.go:92 org_id=fake latency=fast >
Jun 06 15:53:11 dvviows00b0000p loki[990]: ts=2022-06-06T13:53:11.895304144Z caller=spanlogger.go:87 org_id=fake Ingester.TotalReached>
Jun 06 15:53:11 dvviows00b0000p loki[990]: ts=2022-06-06T13:53:11.895340529Z caller=spanlogger.go:87 org_id=fake Summary.BytesProcesse>
Jun 06 15:53:11 dvviows00b0000p loki[990]: level=info ts=2022-06-06T13:53:11.895425553Z caller=metrics.go:92 org_id=fake latency=fast >
lines 1-19/19 (END)
```

Figure 3.7: Running Loki

Setting up Grafana

We also configured a systemd unit file for Grafana that allowed us to run it as a service. We can see in the following figure that it's running successfully:

```
root@dvviows00b0000p:~# systemctl status grafana-server.service
● grafana-server.service - Grafana instance
     Loaded: loaded (/lib/systemd/system/grafana-server.service; enabled; vendor preset: enabled)
     Active: active (running) since Mon 2022-05-16 16:04:21 CEST; 2 weeks 6 days ago
       Docs: http://docs.grafana.org
   Main PID: 1204 (grafana-server)
      Tasks: 12 (limit: 4576)
     Memory: 156.8M
     CGroup: /system.slice/grafana-server.service
             └─1204 /usr/sbin/grafana-server --config=/etc/grafana/grafana.ini --pidfile=/run/grafana/grafana-server.pid -

Jun 06 15:52:03 dvviows00b0000p grafana-server[1204]: logger=ngalert t=2022-06-06T15:52:03.43+0200 lvl=eror msg="error in >
Jun 06 15:52:04 dvviows00b0000p grafana-server[1204]: client.go:172: http://10.106.83.105:3100/loki/api/v1/query_range?dir>
Jun 06 15:52:04 dvviows00b0000p grafana-server[1204]: logger=ngalert t=2022-06-06T15:52:04.19+0200 lvl=eror msg="error in >
Jun 06 15:52:04 dvviows00b0000p grafana-server[1204]: client.go:172: http://10.106.83.105:3100/loki/api/v1/query_range?dir>
Jun 06 15:52:05 dvviows00b0000p grafana-server[1204]: client.go:172: http://10.106.83.105:3100/loki/api/v1/query_range?dir>
Jun 06 15:52:05 dvviows00b0000p grafana-server[1204]: logger=alertmanager org=3 level=debug component=dispatcher msg="Rece>
Jun 06 15:52:06 dvviows00b0000p grafana-server[1204]: client.go:172: http://10.106.83.105:3100/loki/api/v1/query_range?dir>
Jun 06 15:52:06 dvviows00b0000p grafana-server[1204]: logger=alertmanager org=3 level=debug component=dispatcher msg="Rece>
Jun 06 15:52:07 dvviows00b0000p grafana-server[1204]: client.go:172: http://10.106.83.105:3100/loki/api/v1/query_range?dir>
Jun 06 15:52:07 dvviows00b0000p grafana-server[1204]: logger=alertmanager org=3 level=debug component=dispatcher msg="Rece>
```

Figure 3.8: Running Grafana

Setting up Alertmanager

Finally, we configured Alertmanager. We can see in the following figure that it's running successfully:

```
root@dvviows00b0000p:/opt/prometheus# systemctl status alertmanager.service
● alertmanager.service - Alertmanager for prometheus
     Loaded: loaded (/etc/systemd/system/alertmanager.service; enabled; vendor preset: enabled)
     Active: active (running) since Sun 2022-06-12 19:35:04 CEST; 24s ago
   Main PID: 1102015 (alertmanager)
      Tasks: 8 (limit: 4576)
     Memory: 13.4M
     CGroup: /system.slice/alertmanager.service
             └─1102015 /usr/local/bin/alertmanager/alertmanager --config.file=/usr/local/bin/alertmanager/alertmanager.yml --s>

Jun 12 19:35:04 dvviows00b0000p alertmanager[1102015]: ts=2022-06-12T17:35:04.347Z caller=main.go:232 level=info build_context>
Jun 12 19:35:04 dvviows00b0000p alertmanager[1102015]: ts=2022-06-12T17:35:04.348Z caller=cluster.go:185 level=info component=>
Jun 12 19:35:04 dvviows00b0000p alertmanager[1102015]: ts=2022-06-12T17:35:04.353Z caller=cluster.go:680 level=info component=>
Jun 12 19:35:04 dvviows00b0000p alertmanager[1102015]: ts=2022-06-12T17:35:04.402Z caller=coordinator.go:113 level=info compon>
Jun 12 19:35:04 dvviows00b0000p alertmanager[1102015]: ts=2022-06-12T17:35:04.402Z caller=coordinator.go:126 level=info compon>
Jun 12 19:35:04 dvviows00b0000p alertmanager[1102015]: ts=2022-06-12T17:35:04.406Z caller=main.go:431 level=info component=con>
Jun 12 19:35:04 dvviows00b0000p alertmanager[1102015]: ts=2022-06-12T17:35:04.407Z caller=main.go:535 level=info msg=Listening>
Jun 12 19:35:06 dvviows00b0000p alertmanager[1102015]: ts=2022-06-12T17:35:06.354Z caller=cluster.go:705 level=info component=>
Jun 12 19:35:14 dvviows00b0000p alertmanager[1102015]: ts=2022-06-12T17:35:14.356Z caller=cluster.go:697 level=info component=>
```

Figure 3.9: Running Alertmanager

3.3 System Monitoring

In order to implement system monitoring within our proposed solution, we need to configure Node Exporter to expose the hardware and operating system on each VM that we want to monitor.

3.3.1 Setting up Node exporter

First, we start by creating a systemd service for node exporter based on its binary as shown in the screenshot below. We also activated the systemd collector tag in order to expose additional metrics related to systemd running services.

```
[root@dvviows00b0000i opt]# cat /etc/systemd/system/node-exporter.service
[Unit]
Description=Prometheus exporter for machine metrics

[Service]
Restart=always
ExecStart=/usr/local/bin/node_exporter --collector.systemd
ExecReload=/bin/kill -HUP $MAINPID
TimeoutStopSec=20s
SendSIGKILL=no

[Install]
WantedBy=multi-user.target
```

Figure 3.10: Node exporter systemd Unit file

The next step is to add the new target to the Prometheus configuration file prometheus.yml, so that the desired metrics are pulled. Figure 3.11 shows how a new target is configured. In fact, for each target, we added specific labels (group/env) so we can later distinguish between different VMs in different environments.

```
- job_name: 'Fop_qualif1_was1'
  scrape_interval: 10s
  static_configs:
    - targets: ['10.118.12.166:9100']
      labels:
       group: Fop
       env: q1
```

Figure 3.11: Node exporter systemd Unit file

After configuring the Node exporter, the VM's metrics are now exposed through the URL: IPaddress:9100/metrics as shown in the figure below :

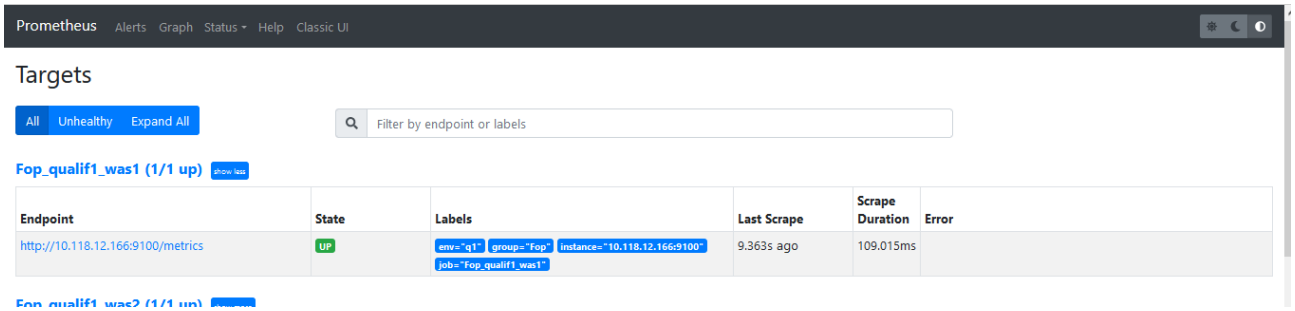

Figure 3.12: Node exporter successfully monitored by Prometheus

```
[root@dvviows00b0000i ~]# curl 10.118.12.166:9100/metrics
# 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"} 2.5891e-05
go_gc_duration_seconds{quantile="0.25"} 4.6227e-05
go_gc_duration_seconds{quantile="0.5"} 4.8615e-05
go_gc_duration_seconds{quantile="0.75"} 5.1579e-05
go_gc_duration_seconds{quantile="1"} 0.00033115
go_gc_duration_seconds_sum 95.498495027
go_gc_duration_seconds_count 2.027254e+06
# HELP go_goroutines Number of goroutines that currently exist.
# TYPE go_goroutines gauge
go_goroutines 9
# HELP go_info Information about the Go environment.
# TYPE go_info gauge
go_info{version="go1.17.3"} 1
# HELP go_memstats_alloc_bytes Number of bytes allocated and still in use.
# TYPE go_memstats_alloc_bytes gauge
go_memstats_alloc_bytes 2.604216e+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 5.16171401508e+12
# 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 2.463151e+06
# HELP go_memstats_frees_total Total number of frees.
# TYPE go_memstats_frees_total counter
go_memstats_frees_total 1.18982234005e+11
# HELP go_memstats_gc_cpu_fraction The fraction of this program's available CPU time used by the GC since the program started.
# TYPE go_memstats_gc_cpu_fraction gauge
go_memstats_gc_cpu_fraction 0.0008986512761667383
# HELP go_memstats_gc_sys_bytes Number of bytes used for garbage collection system metadata.
# TYPE go_memstats_gc_sys_bytes gauge
```

Figure 3.13: Metrics exposed by Node exporter

Now that we are successfully pulling metrics, we move forward to querying these metrics through the Grafana server to turn them into significant and appealing graphs.
Starting by system-related metrics: **Memory metrics**
Instead of running Linux commands such as "free", which gives information about memory state, we are going to make use of Node Exporter metrics related to memory usage. In fact, we focused on the following metrics:

- node-memory-MemTotal-bytes

- node-memory-MemFree-bytes
- node-memory-Buffers-byte
- node-memory-Cached-bytes
- node-memory-SwapTotal-byte
- node-memory-SwapFree-bytes

Computing the total RAM used is equivalent to using node-memory-MemTotal-bytes for each VM, as shown in the picture below:

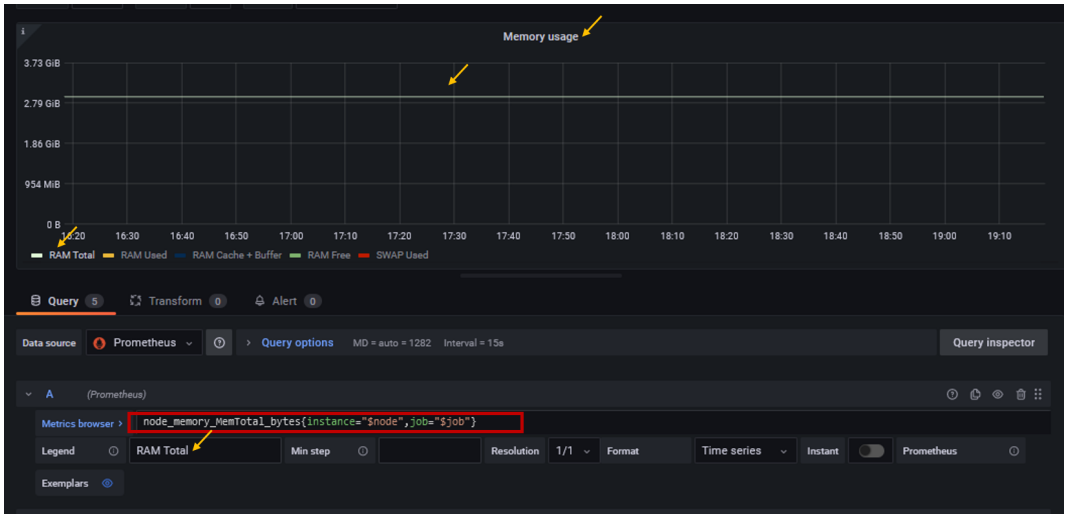

Figure 3.14: Total memory available

Prometheus scrapes its targets every 5 seconds, thus, each expression is computed every 5 seconds. The table below shows how we compute different memory-related metrics :

Value	Expression
RAM used	node-memory-MemTotal-bytes - node-memory-MemFree-bytes - node-memory-Buffers-bytes - node-memory-Cached-bytes
RAM cache+Buffer	node-memory-Buffers-bytes + node-memory-Cached-bytes
SWAP used	- node-memory-SwapTotal-bytes - node-memory-SwapFree-bytes
Free RAM	node-memory-MemFree-bytes

Table 3.1: Memory metric expressions

And the final output is as shown in the picture below :

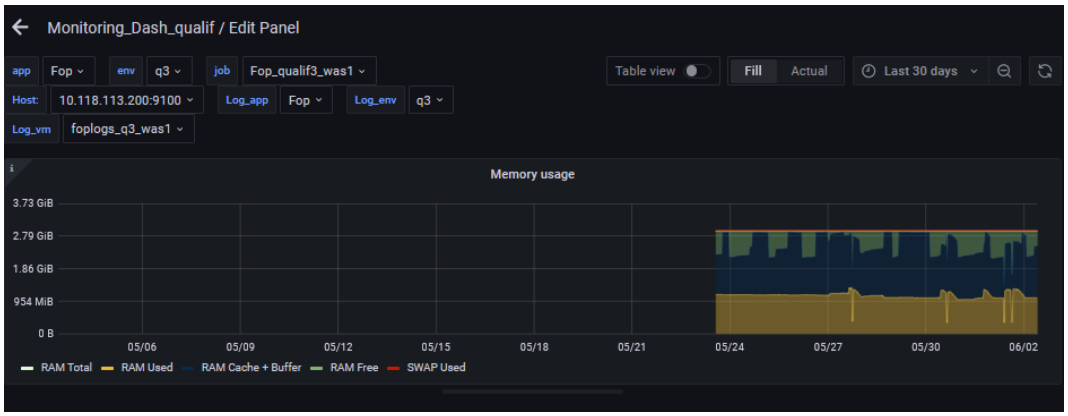

Figure 3.15: Memory usage graph

Disk metrics

To compute the percentage of disk usage we used the following expressions :
100 - (node-filesystem-avail-bytes/node-filesystem-size-bytes * 100).

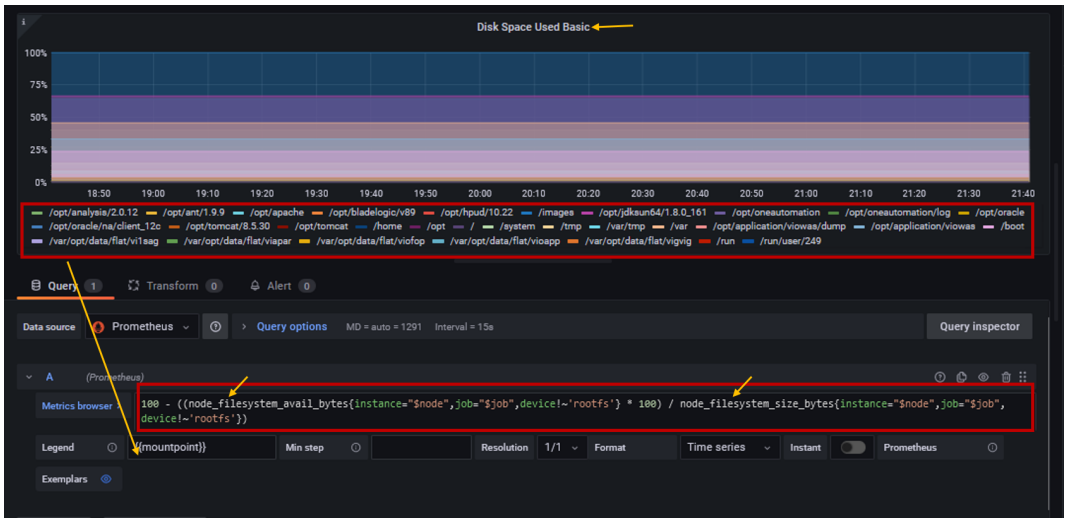

Figure 3.16: Percentage of Disk space used

Network metrics

To monitor outgoing and ingoing network traffic we compute the rate of received and transmitted bytes through different network interfaces :

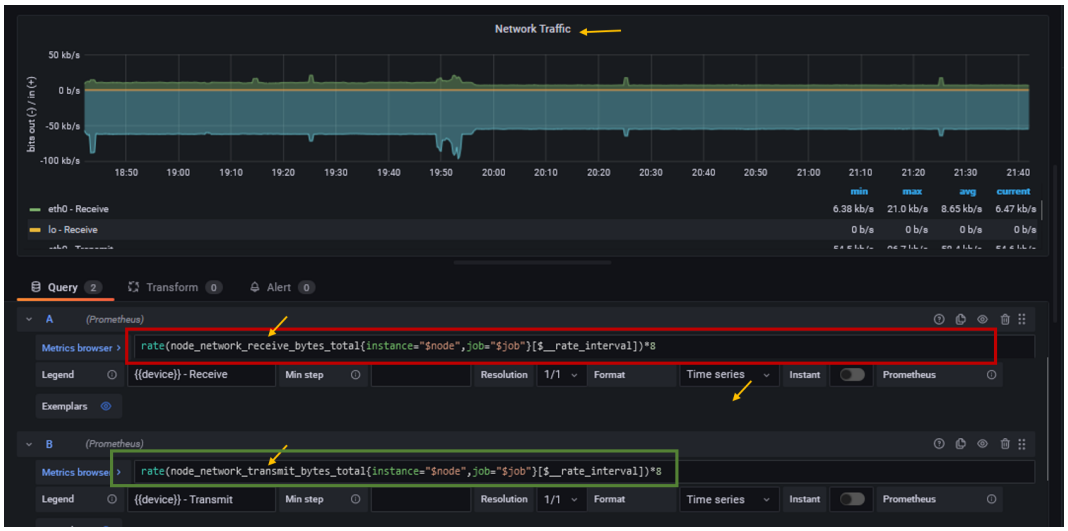

Figure 3.17: Network trafic metric

systemd metrics

By activating the systemd collector previously, we were able to compute the number of systemd services classified by their current state :

Figure 3.18: Number of systemd services by state

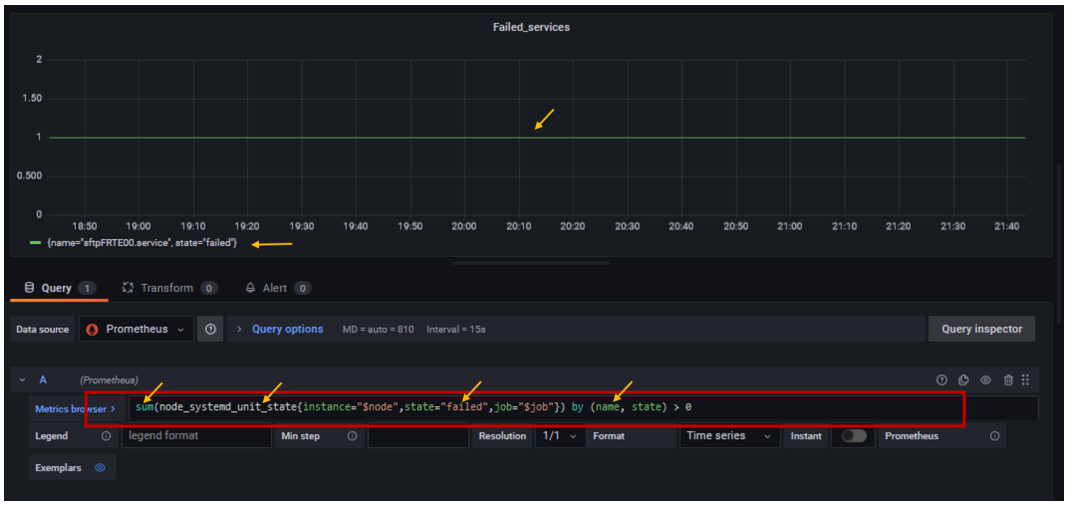

Figure 3.19: Failed systemd services

Metrics collected by cron-jobs

In order to monitor how Prometheus data directory is evolving, we added a cron job that collects data about the Prometheus data directory through a texfile collector, which exposes metrics from files written by other processes. Below is the script running to format the output of the Linux command "du -ms" into a readable Prometheus metric.

```
*/5 * * * * root du -ms /opt/prometheus/data /var/log | sed -ne 's/^\([0-9]\+\)\t\(.*\)$/node_directory_size_bytes{director
y="\2"} \1/p' > /var/lib/node_exporter/textfile_collector/directory_sizes.prom.$$ && mv /var/lib/node_exporter/textfile_col
lector/directory_sizes.prom.$$ /var/lib/node_exporter/textfile_collector/directory_sizes.prom
~
```

Figure 3.20: Cron-job script

3.4 Availability Monitoring

Our solution should ensure the monitoring of applications' availability. To provide this type of supervision, we setup a blackbox exporter as a first step:

Blackbox exporter setup

In order to have a functional blackbox exporter, we modified the configuration file so that our exporter can probe different endpoints through HTTP. In the configuration file below, for any status code different form 200, we will assume that the endpoint is down:

```
root@dvviows00b0000p:~# cat /etc/blackbox/blackbox.yml
modules:
  http_2xx:
    prober: http
    timeout: 5s
    http:
      valid_status_codes: [200]
      method: GET
```

Figure 3.21: Blackbox exporter configuration file

The following step is to add blackbox exporter to the prometheus.yml configuration file:

```
- job_name: 'blackbox'
  scrape_interval: 10s
  metrics_path: /probe
  params:
    module: [http_2xx]
  static_configs:
    - targets:
      - http://dvviows00b00000.com.intraorange:8080/jsp    # Target to probe with https.
  relabel_configs:
    - source_labels: [__address__]
      target_label: __param_target
    - source_labels: [__param_target]
      target_label: instance
    - target_label: __address__
      replacement: 10.106.83.105:9115  # The blackbox exporter's real hostname:port.
```

Figure 3.22: Adding blackbox exporter to prometheus.yml

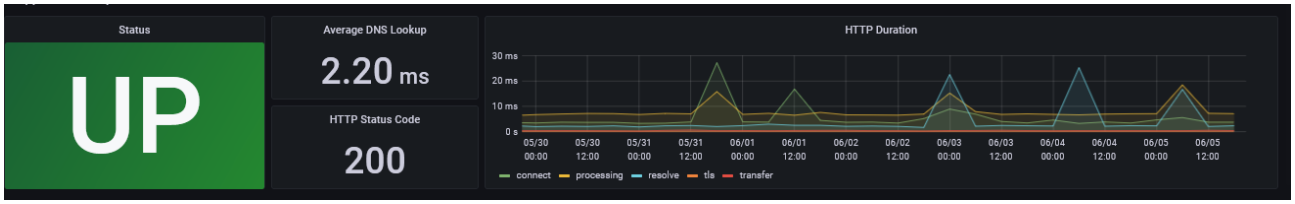

Figure 3.23: Status of application endpoint

3.5 Log Monitoring

Log monitoring is an essential part of our solution. We implemented this concept to supervise error and exception rates within our applications.We first started by setting up the Promtail agent, which will be responsible for shipping log streams to the Loki server through the Loki API.We modified the Promtail configuration file by specifying the path of log files and adding specific labels to distinguish between their different sources.

```
[root@dvviows00b0000i ~]# cat /opt/promtail/promtail-local-config.yaml
server:
  http_listen_port: 9080
  grpc_listen_port: 0

positions:
  filename: /tmp/positions.yaml

clients:
  - url: http://10.106.83.105:3100/loki/api/v1/push
scrape_configs:
- job_name: log
  static_configs:
  - targets:
      - localhost
    labels:
      host: foplogs_q3_was3
      env:  q3
      app:  Fop
      __path__: /opt/application/viowas/current/tomcat/00/logs/{fo_journal,fo_servlet}.log
[root@dvviows00b0000i ~]#
```

Figure 3.24: Promtail configuration file

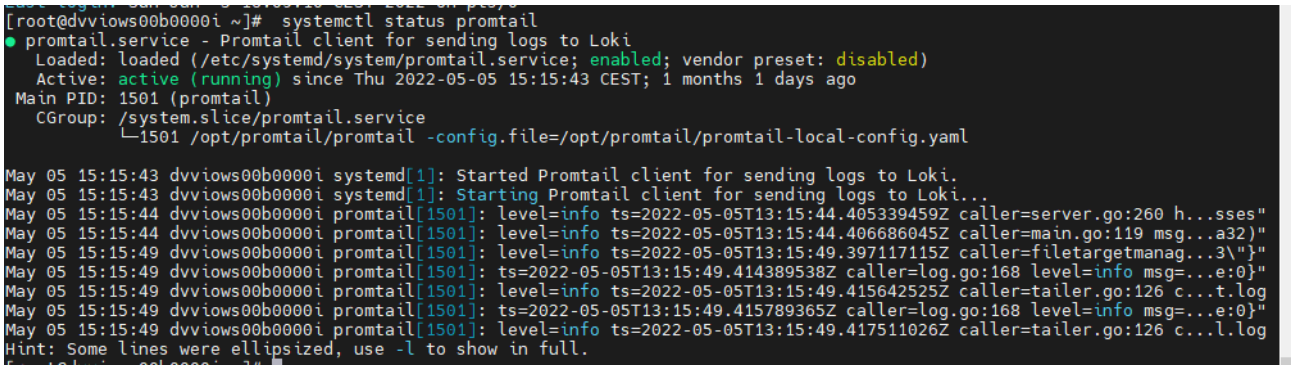

Figure 3.25: Running Promtail status

After successfully receiving logs from different VMs, we can explore log files in Grafana as shown in the screenshot below :

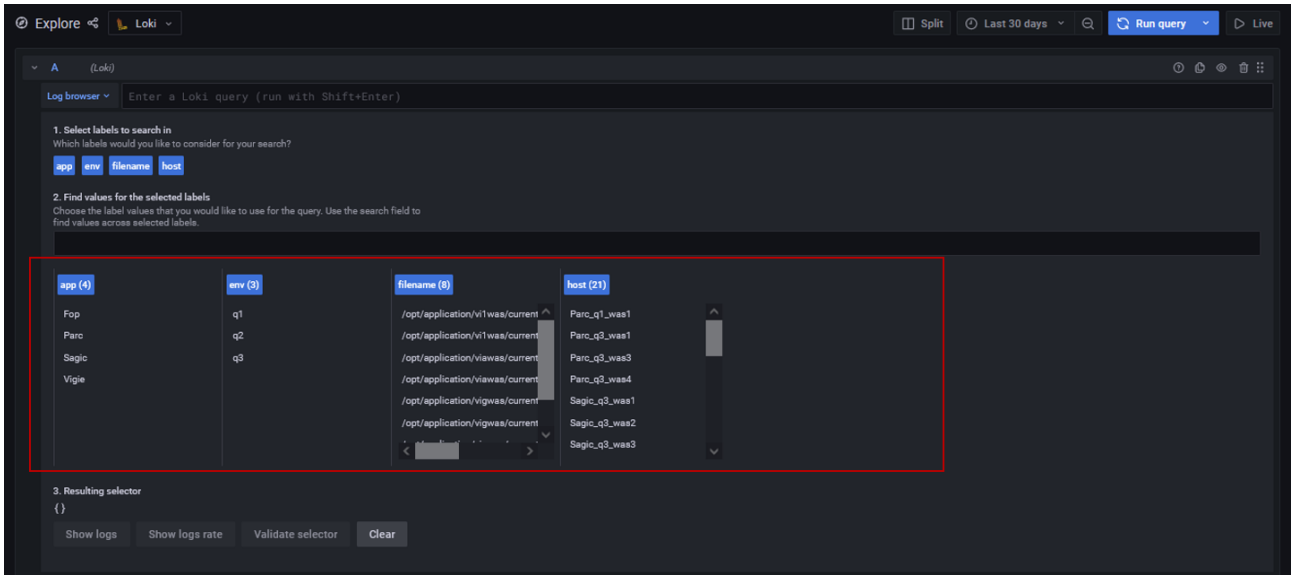

Figure 3.26: Explore Log files

Querying metrics

We derived metrics related to several failures based on the log files we received: We decided not to format the log lines into various parts since the log description might be anything relevant to the application, and there is no expression to fulfill all possibilities, because the same log line description can contain information about the queues, the application, and its database.

Another factor to consider is the scarcity of resources. We must label our log files carefully to avoid overloading our monitoring server, because we are keeping data directly on our virtual machine. As a result, we used labels to designate the source of log files, as described previously. We used LogQL filters to identify distinct types of faults in a log stream.

- General errors :

For general erros we will search for the pattern ”ERREUR” in log streams as shown in the figure below:

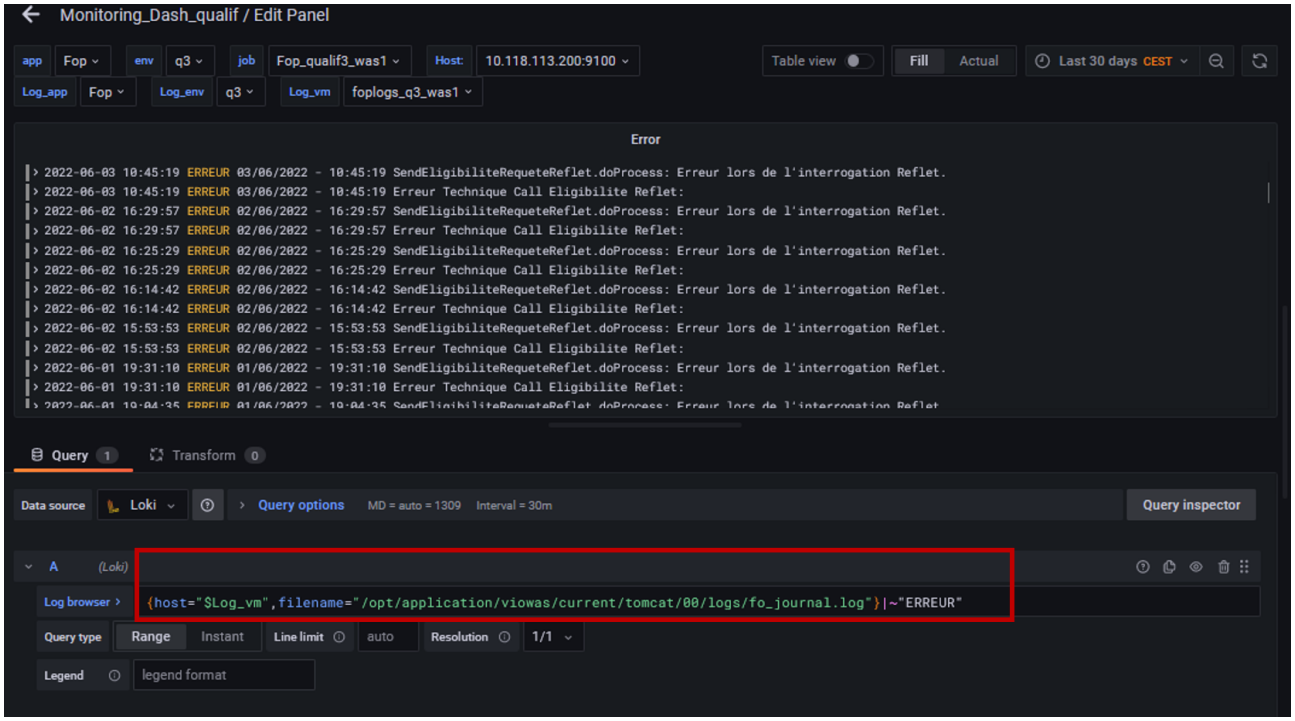

Figure 3.27: filtering error log lines

We extracted patterns that were repeated throughout all log files for each error type, suggesting that a specific problem occurred. Figure 3.28 below show how we identified each error type.

- Java exceptions:

For java exceptions we will search for the pattern ”java.*Exception” in log streams as shown in the figure below:

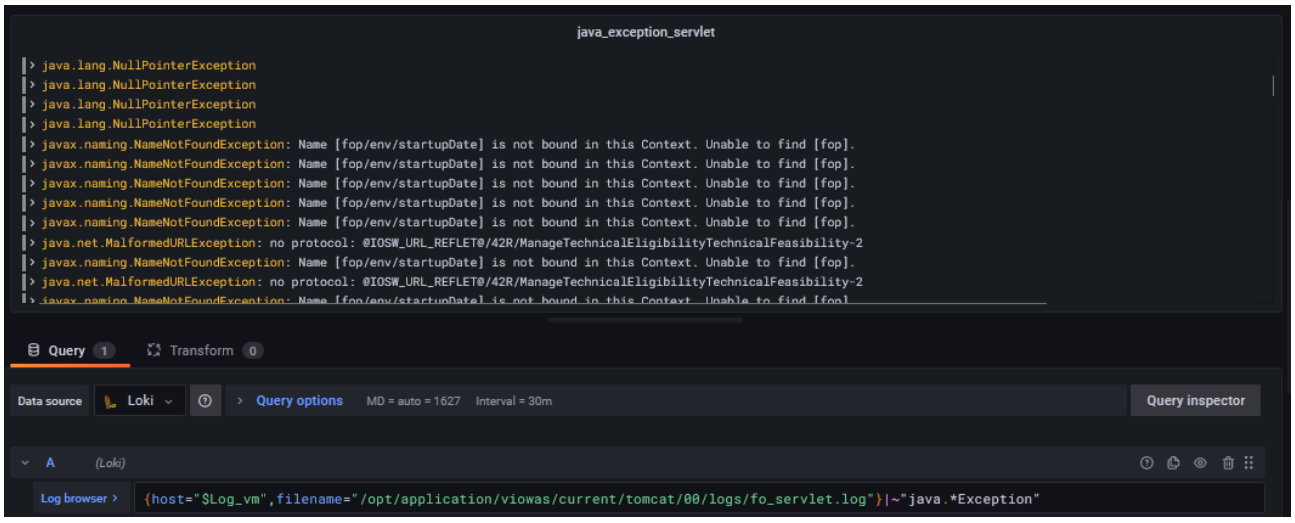

Figure 3.28: filtering log line with java exception

- Database errors :

For DBs errors we will search for the pattern ”ORA-” in log streams as shown in the figure below:

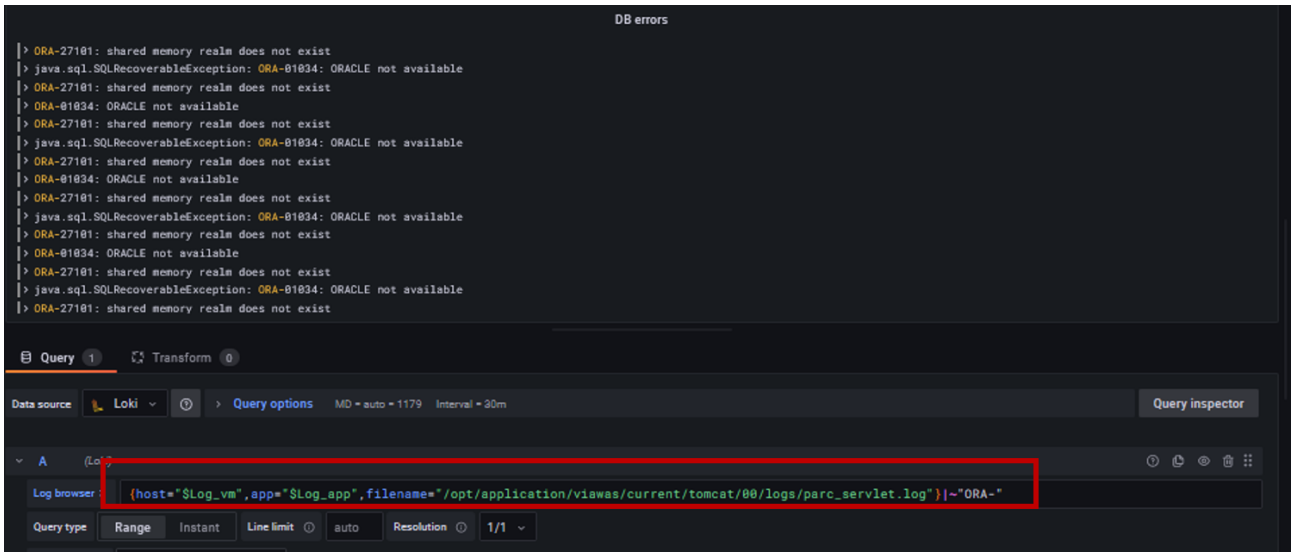

Figure 3.29: filtering log line with Database errors

- Queue errors :

For queue errors we will search for log streams that start with the pattern ”JMS” as shown in the figure below:

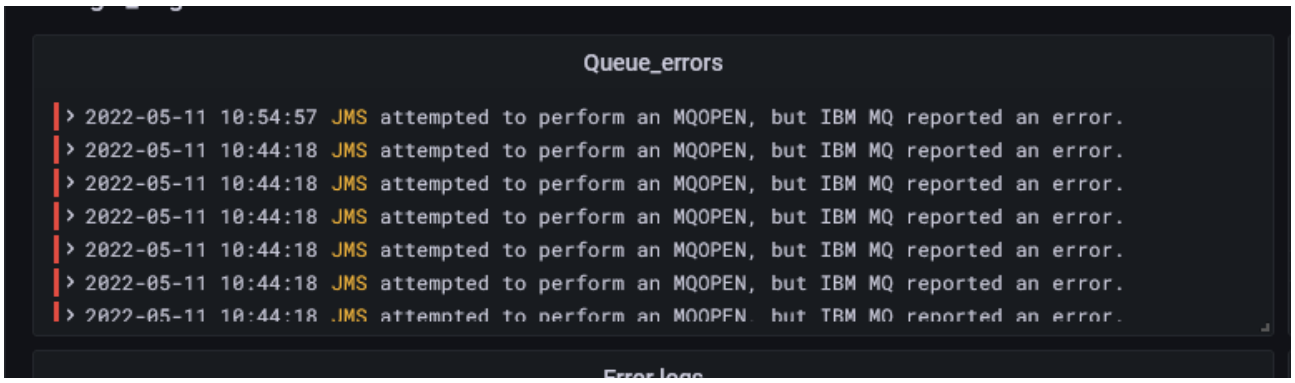

Figure 3.30: filtering log line with Queue errors

In addition, in order to visualize the number of errors through different intervals, we used range vectors such as rate:

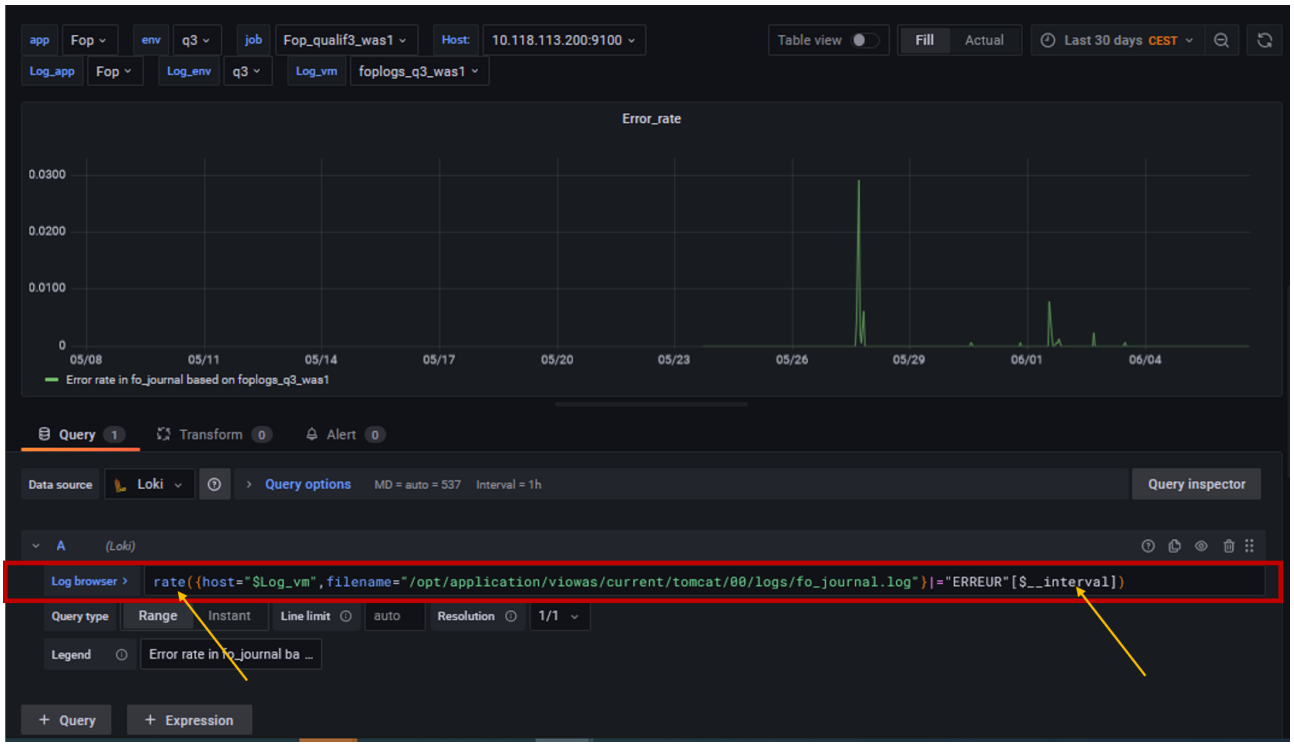

Figure 3.31: The rate of errors through time

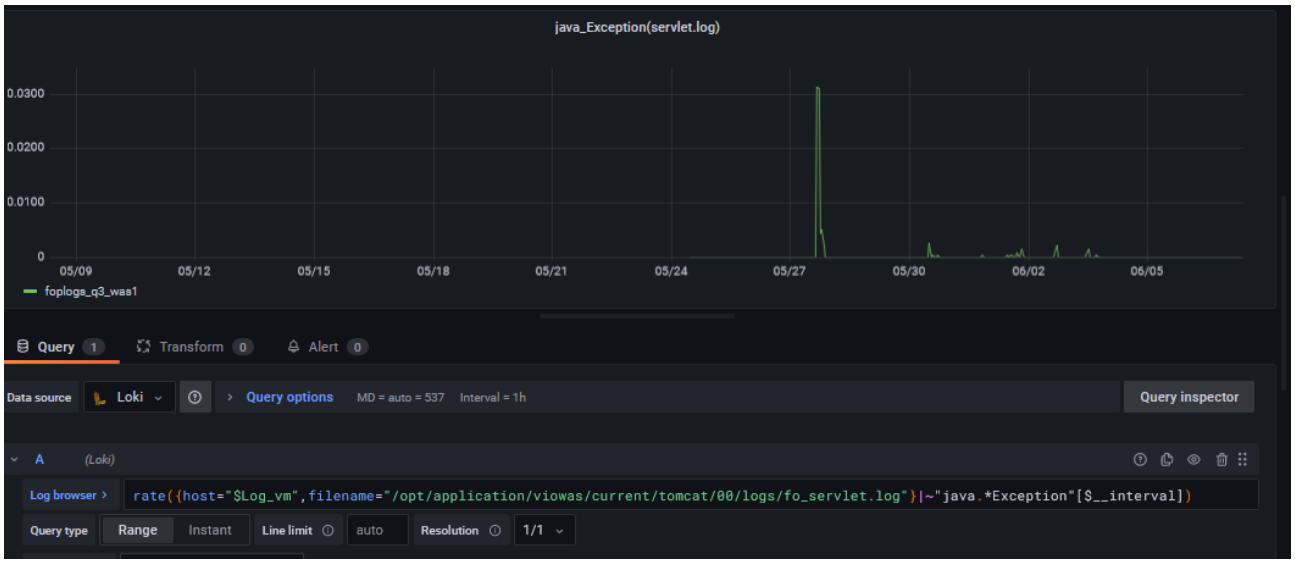

Figure 3.32: The rate of java exceptions through time

To have a closer look on different java exceptions , we counted the ones that usually occurs through the range vector count-over-time:

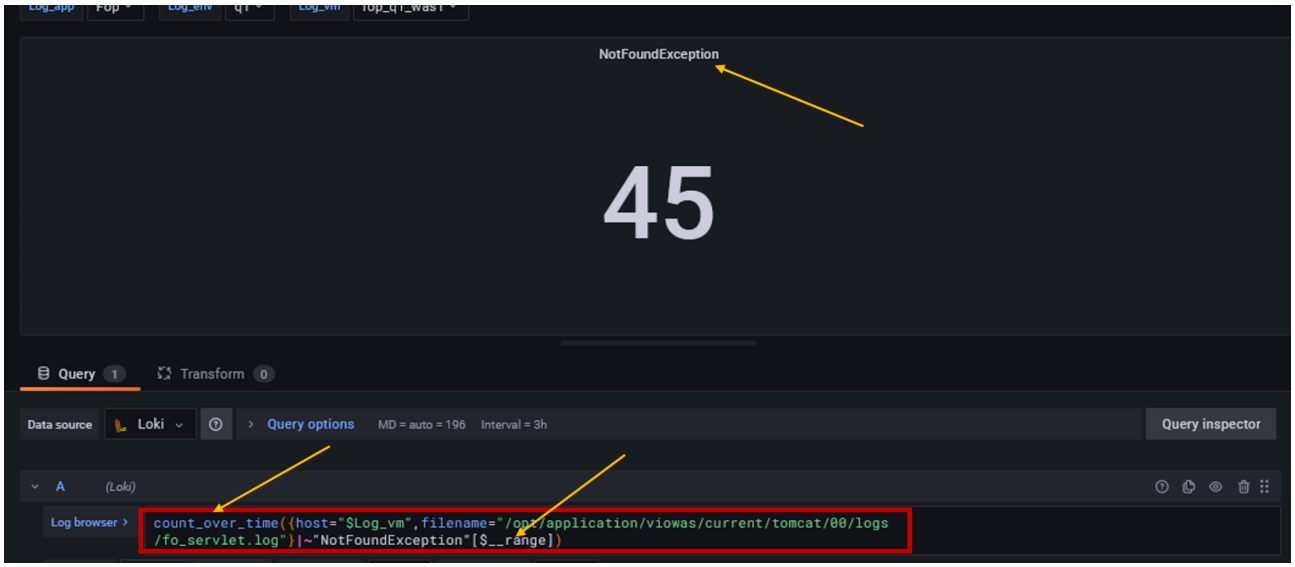

Figure 3.33: Number of "NotFoundExceptions"

Following the same process in the previous figure, we created different panels for each type of exception:

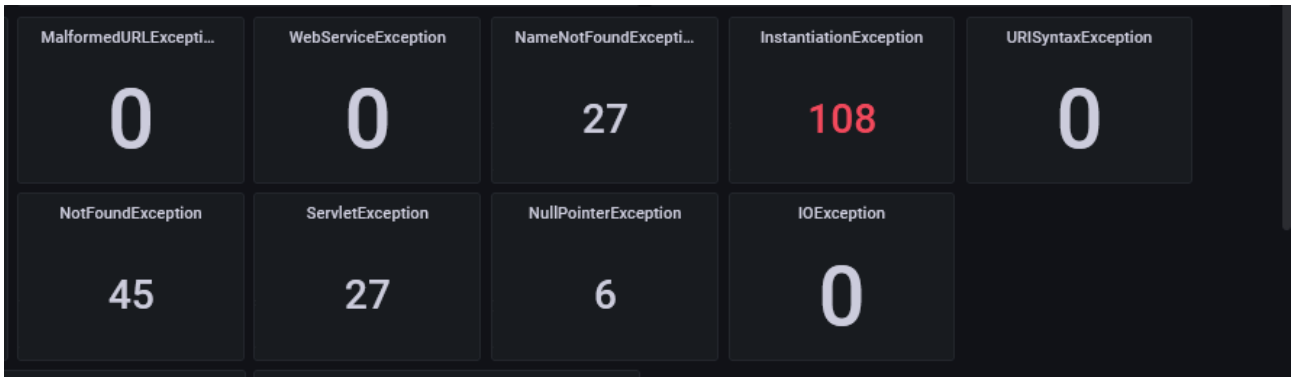

Figure 3.34: Number of exceptions by type

3.6 Variables

To achieve our goal of creating a single dashboard from which we can navigate between dashboards of multiple VMs, we added Grafana variables based on the labels we generated before in Prometheus and Loki.
The variables created for this dashboard are shown in Figure 3.35.

Figure 3.35: Dashboards Variables

To create a variable we based our query on a Prometheus metric node-uname-info we can see in the Figure 3.36 what labels this metric have:

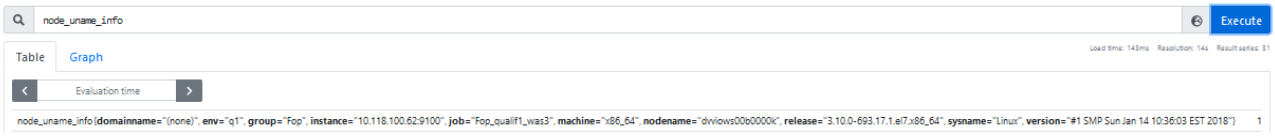

Figure 3.36: Extracting VM name as a variable

In Grafana dashboard we called the metric (node-uname-info) through the function label-values, that allows us to extract labels such as job, application, or environment. Besides, in our case variables depend on each other, that's why we called them through the $ symbol.

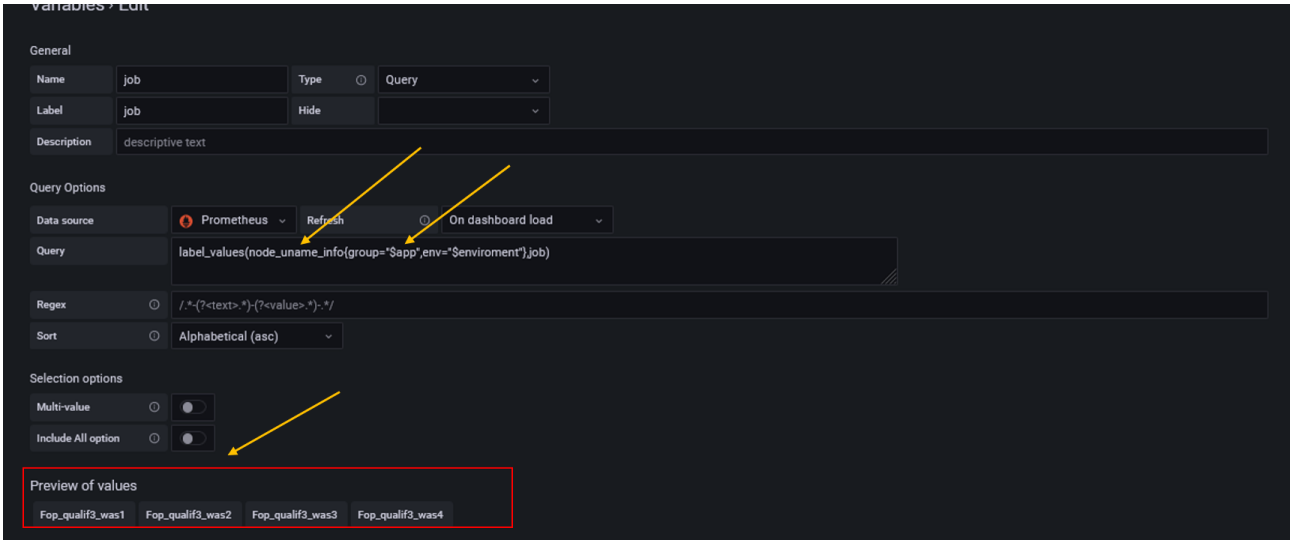

Figure 3.37: Extracting VM name as a variable

These variables allowed us to develop these drop down lists, which allow the user to easily browse between different apps, environments, and virtual machines (VMs):

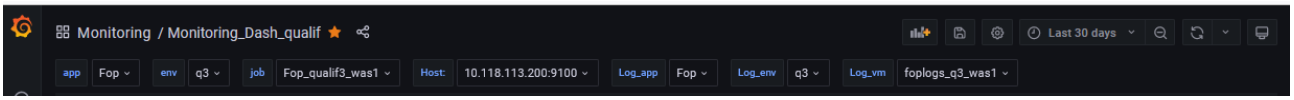

Figure 3.38: Different drop down lists

3.7 Alerts

To add alerting to our dashboard, we created different type of alerts. Alerts can be produced directly in the Grafana dashboard or through Alertmanager.In this project, we used both methods.

Log related Alerts

First, we developed alerts for various types of failures; then, to make it easier to see these warnings, we classified them by environment and application, as shown in the figure below:

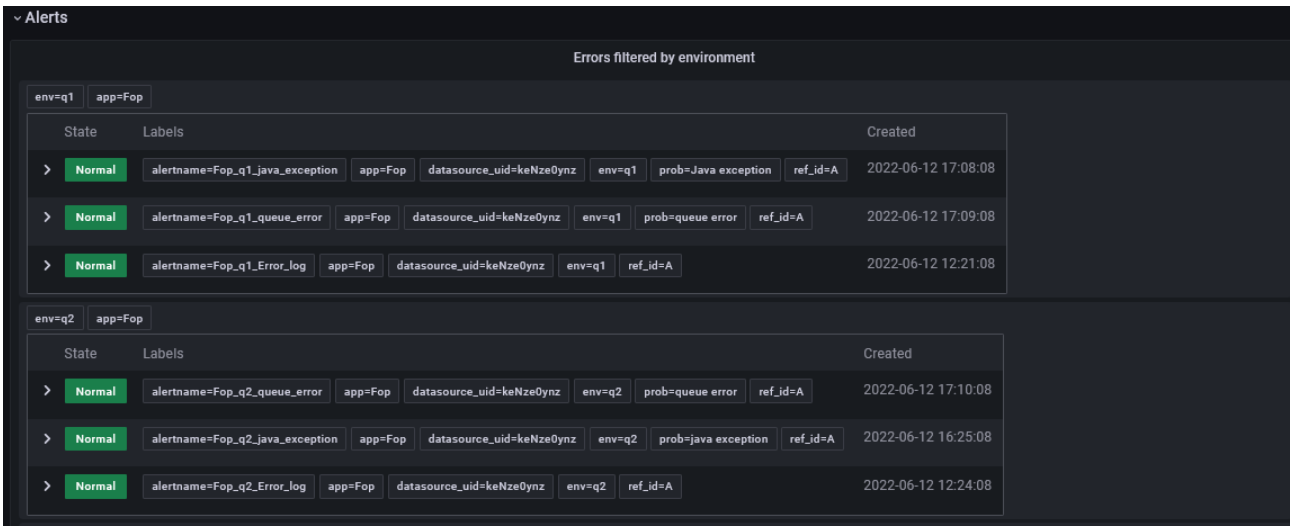

Figure 3.39: Alerts grouped by environment/application

Below is a close up on how we created each alert in Grafana:

First, we used the range vector count-over-time, and then we added a trigger to generate an alert whenever the output of our range vector exceeds 0. Finally, the alert will be evaluated every 1 minute for 2 minutes.

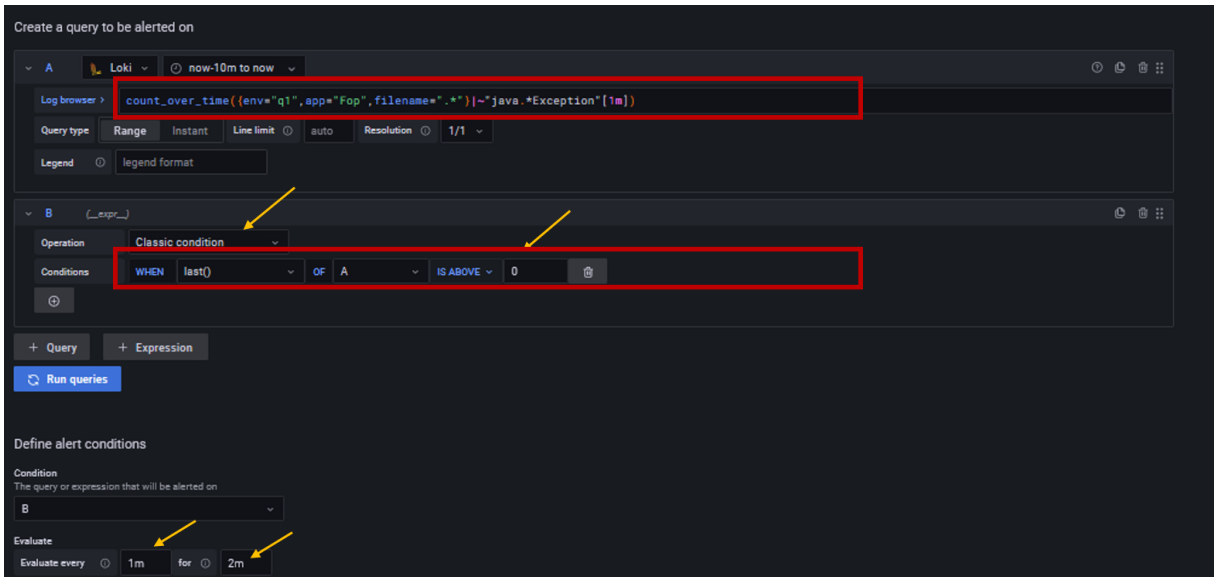

Figure 3.40: Alert for a java exception

System related Alerts

We set an alert for each time a Prometheus instance goes down, as well as another alert to monitor the state of Promtail to ensure that logs are being delivered correctly. We also set up a notification for down app endpoints:
The figure below illustrates how we created these alerts through Alertmanager.

```
 GNU nano 4.8                                rules.yml                                Modified
groups:
  - name: test
    rules:
    - alert: InstanceDown
      expr: up == 0
      for: 1m
      labels:
        severity: critical
      annotations:
        summary: A  target missing (instance {{ $labels.instance }})
        description: "A  target has disappeared. An exporter might be crashed.\n  VALUE = {{ $value }}\n  LABELS = {{ $labels }}
    - alert: Promtail service is down
      expr: node_systemd_unit_state{state="failed",name="promtail.service"} == 1
      for: 1m
      labels:
        severity: critical
      annotations:
        summary: Promtail is down in the vm  {{ $labels.job }}
        description: "promtail service is down logs are coming from the machine {{ $labels.job }} are no longer available"
    - alert: Endpointdown
      expr: probe_http_status_code != 200
      for: 1m
      labels:
        severity: critical
      annotations:
        summary: App endpoint is down
        description: "App endpoint is down  {{ $labels.instance }}"
```

Figure 3.41: Rules.yml script

First we define the name of the alert. Second We define, the expressions used, followed by the evaluation time, and the severity level. Finally we add a summary and a description to describe the problem.

The following figure shows how they appear on the dahsboard:

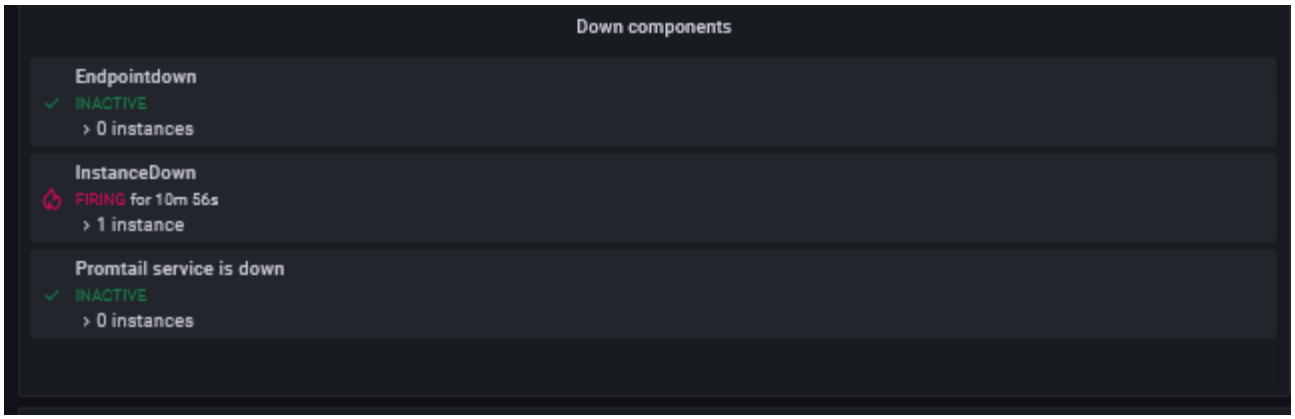

Figure 3.42: Down service/instance

For High usage : we created an alert for High Ram usage as shown in the picture below : We also set up an alert to notify us if our disk space will be filled in 4 hours, using Prometheus' "predict-linear" forecasting tool. The function will be computed every 1h as shown in the figures below:

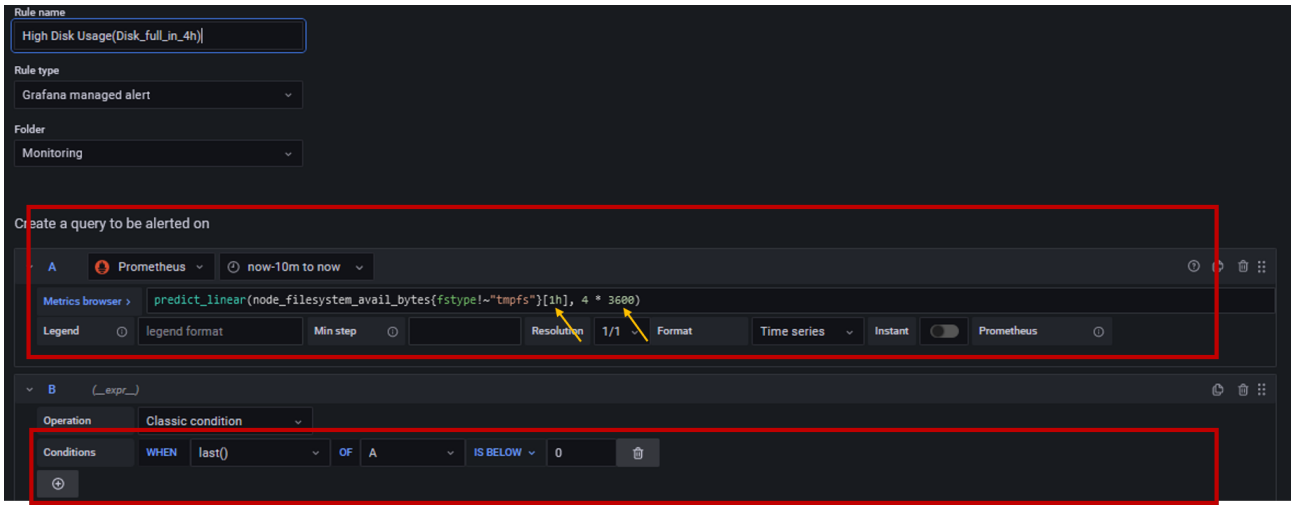

Figure 3.43: Predict-linear forecasting function

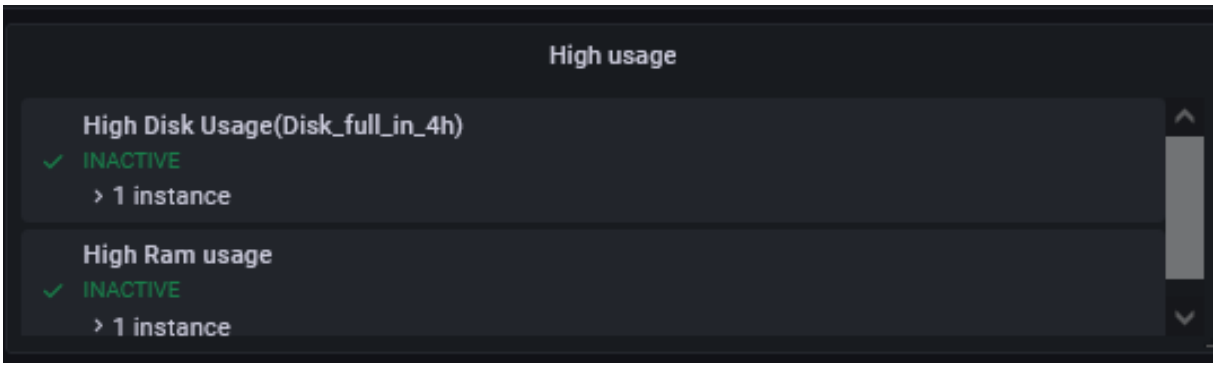

Figure 3.44: High usage alerts

3.8 Automation of adding a new target

To make the process of adding a new target to monitor easier, we used ODE through Gitlab-CI, as explained previously, to automate the installation and configuration of Node exporter and Promtail using the same labels. Once we launch our pipeline and run the playbooks, our target will be added automatically to the dashboard.
Below, are figures that show the successful state of deployment pipelines. Each playbook will, run a set of commands on the target machine:

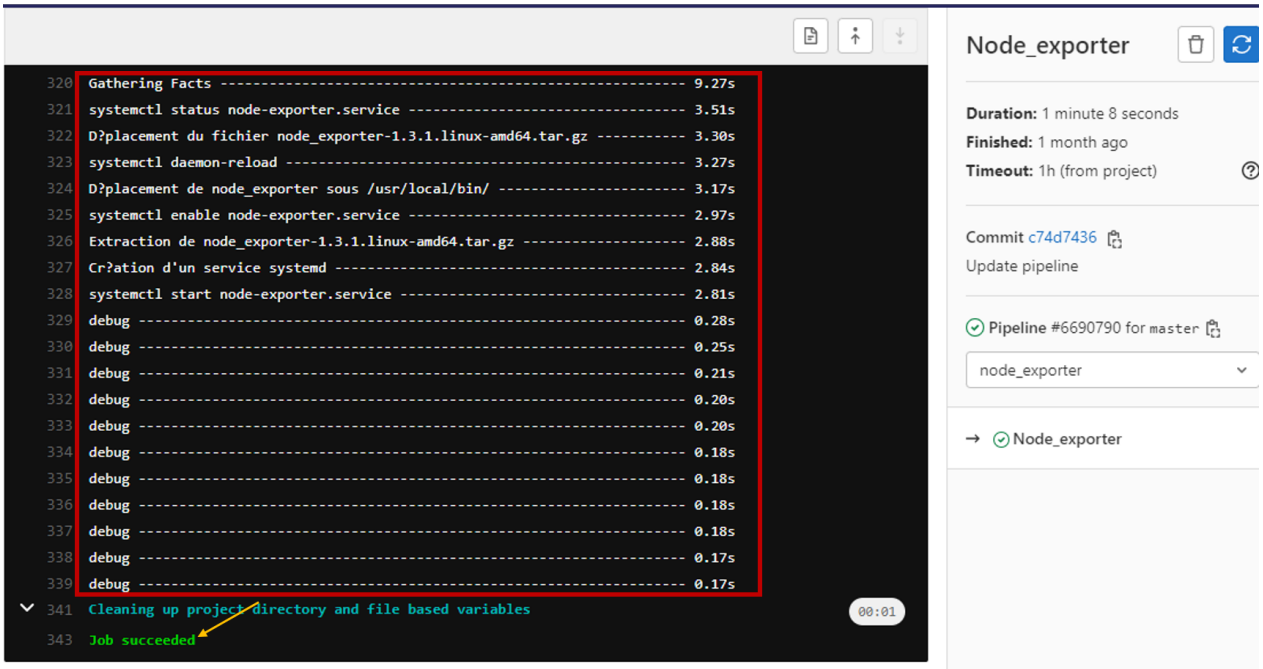

Figure 3.45: Successfully automating the installation and configuration of Node exporter on Gitlab

We had to use Jinja2 template for Promtail so that we could dynamically add labels. Jinja2 templates are basic text files with variables that are substituted by values defined during the Ansible template module's execution. These data can be found in variables

defined in Ansible inventory. In our situation, we used our inventory to define our variables (labels). [19]
The screenshots below illustrate how we named and defined our variables in our template.

```
[all]
#PARC QUA1 WAS1
10.118.12.157 env_host=Parc_q1_was1 env_var=q1 env_app=Parc
#PARC QUA1 WAS2
10.118.12.91 env_host=Parc_q1_was2 env_var=q1 env_app=Parc
#PARC QUA1 WAS3
10.118.118.35 env_host=Parc_q1_was3 env_var=q1 env_app=Parc
#PARC QUA1 WAS4
10.118.63.15 env_host=Parc_q1_was4 env_var=q1 env_app=Parc
#PARC QUA2 WAS1
10.118.48.165 env_host=parc_q2_was1 env_var=q2 env_app=Parc
#PARC QUA3 WAS1
10.118.112.96 env_host=Parc_q3_was1 env_var=q3 env_app=Parc
#PARC QUA3 WAS2
10.118.98.63 env_host=Parc_q3_was2 env_var=q3 env_app=Parc
#PARC QUA3 WAS3
10.118.13.36 env_host=Parc_q3_was3 env_var=q3 env_app=Parc
#PARC QUA3 WAS4
10.118.101.144 env_host=Parc_q3_was4 env_var=q3 env_app=Parc
#FOP QUA1 WAS1
#10.118.12.166 env_host=fop_q1_was1 env_var=q1 env_app=Fop
#FOP QUA1 WAS2
#10.118.13.16 env_host=fop_q1_was2 env_var=q1 env_app=Fop
#FOP QUA1 WAS3
10.118.100.62 env_host=fop_q1_was3 env_var=q1 env_app=Fop
#FOP QUA1 WAS4
10.118.48.33 env_host=fop_q1_was4 env_var=q1 env_app=Fop
#FOP QUA2 WAS1
10.118.55.252 env_host=fop_q2_was1 env_var=q2 env_app=Fop
```

Figure 3.46: Defining labels as variables in the inventory

```
server:
  http_listen_port: 9080
  grpc_listen_port: 0

positions:
  filename: /tmp/positions.yaml

clients:
  - url: http://10.233.96.84:3100/loki/api/v1/push
scrape_configs:
- job_name: log
  static_configs:
  - targets:
      - localhost
    labels:
      host: {{env_host}}
      env:  {{env_var}}
      app:  {{env_app}}
      __path__: /opt/application/viowas/current/tomcat/00/logs/{fo_journal,fo_servlet}.log
```

Figure 3.47: Promtail Jinja2 template

The following figure shows a successful installation of Promtail agent:

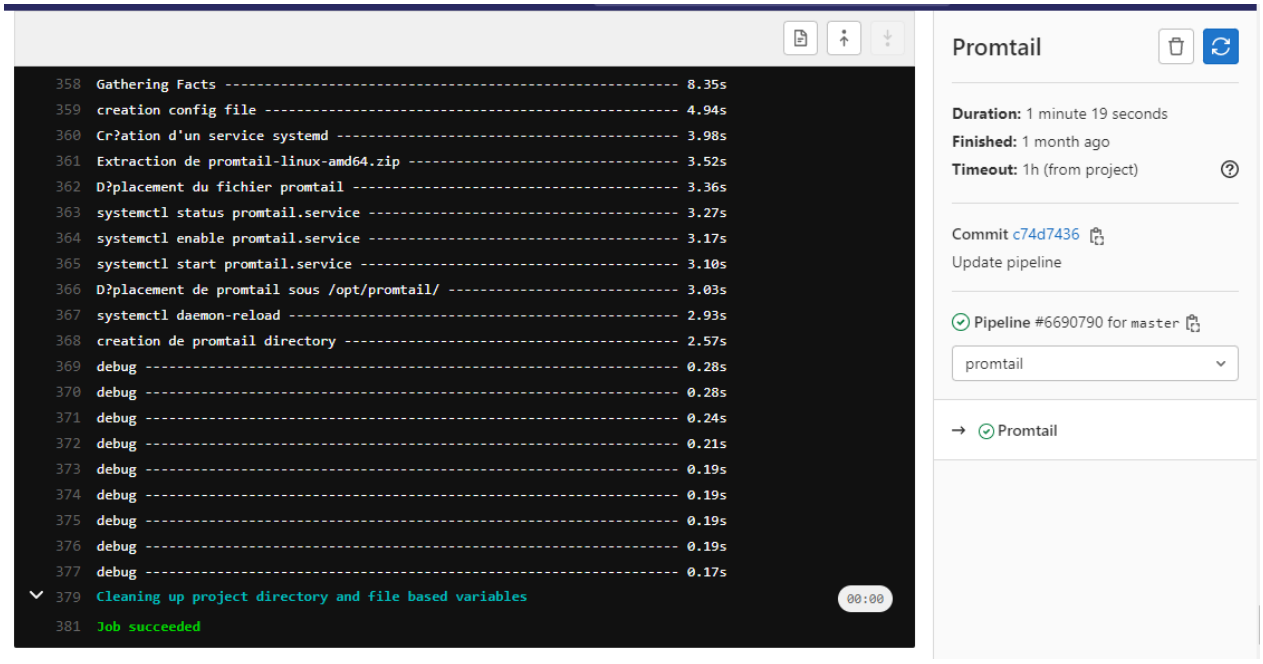

Figure 3.48: Successfully automating the installation and configuration of Promtail on Gitlab

3.9 Automation of adding a new Monitoring instance

We prepared playbooks to automate the installation of a new monitoring stack in the pre-production and production environments to make it easier to deploy a new monitoring instance. As previously described, we began by installing and configuring node exporters and Promtail, and then added targets dynamically once the Prometheus+Grafana+Loki playbooks were launched. The following figure shows a successful installation of Promtheus server:

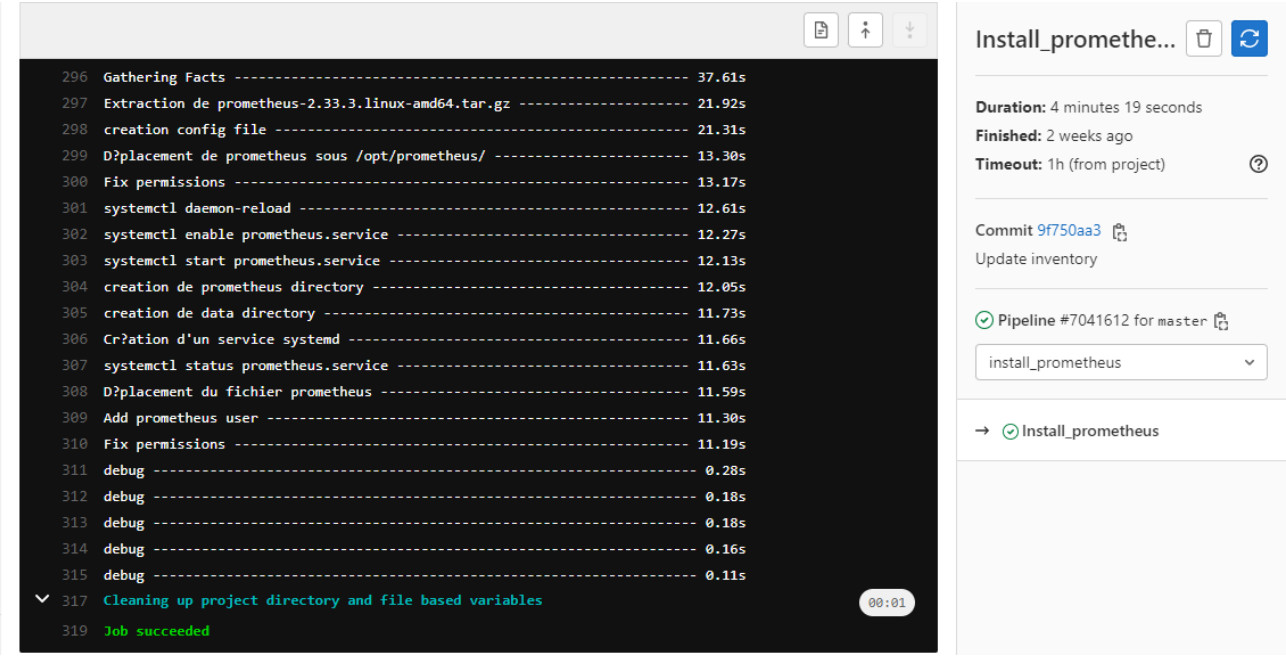

Figure 3.49: Successfully automating the installation and configuration of Promtheus on Gitlab

The following figure shows a successful installation of Loki server:

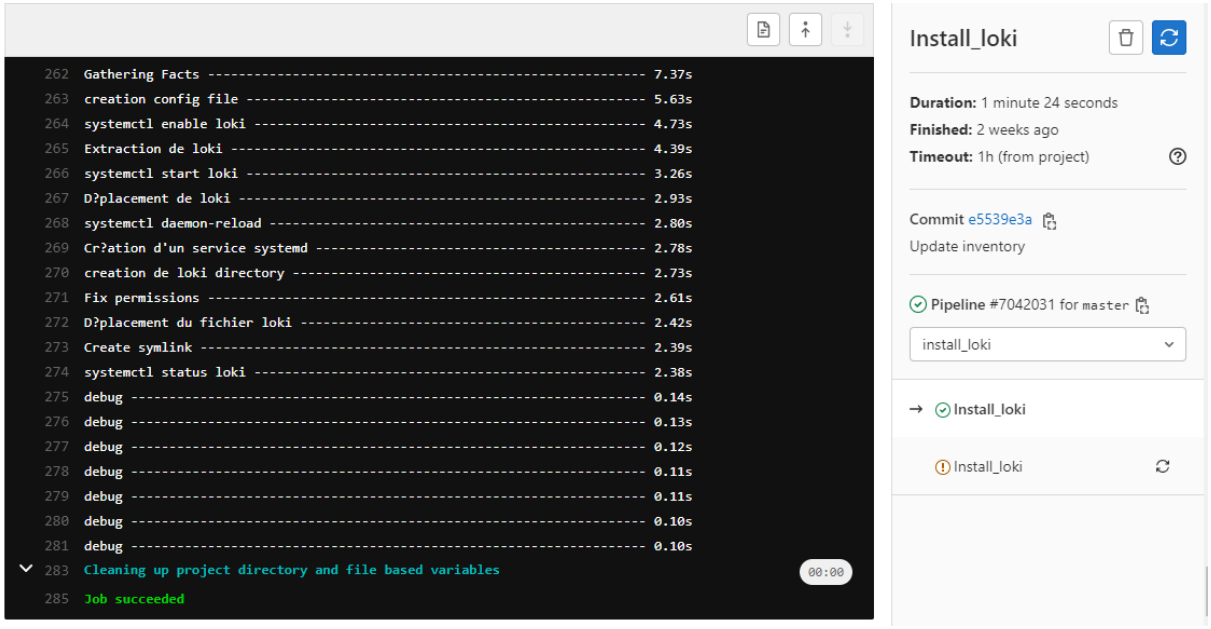

Figure 3.50: Successfully automating the installation and configuration of Loki on Gitlab

The following figure shows a successful installation of Grafana server:

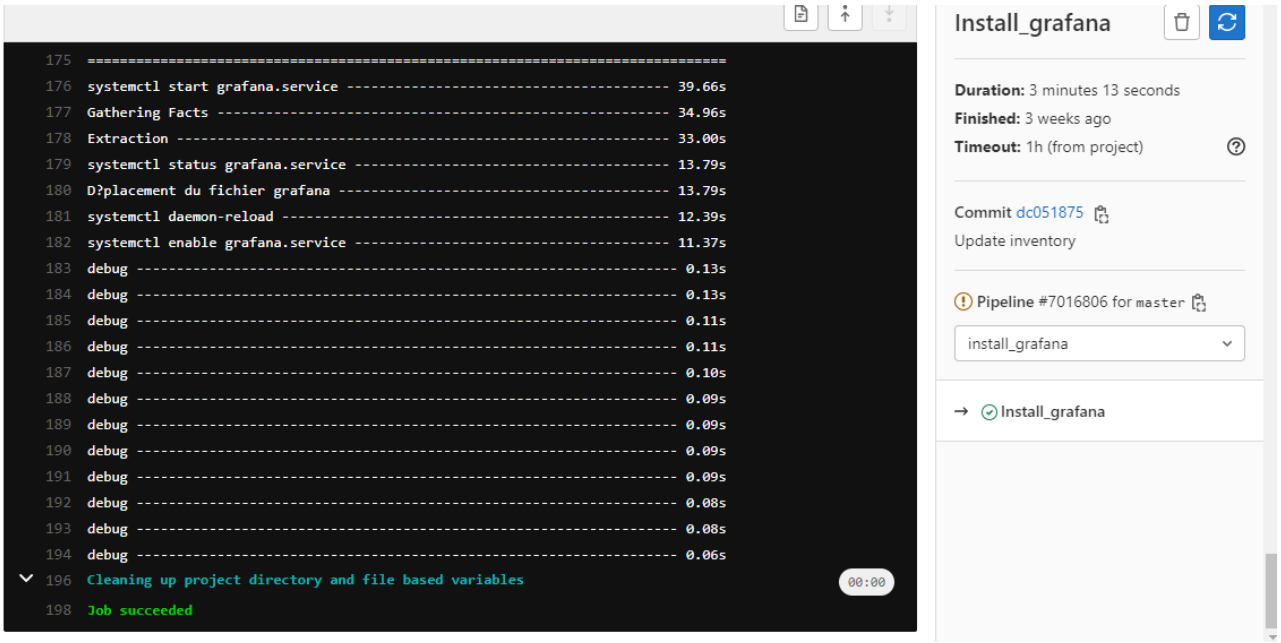

Figure 3.51: Successfully automating the installation and configuration of Grafana on Gitlab

3.10 Testing Metric collection

To test if our metrics are computed in the right way, we based our test on memory metric: the following Figures show that metrics are the same as the output of linux command Free -h :

```
[root@dvviows00b0000j ~]# free -h
              total        used        free      shared  buff/cache   available
Mem:           2.9G        145M        210M        1.0M        2.6G        2.7G
Swap:          1.0G          0B        1.0G
```

Figure 3.52: Free command

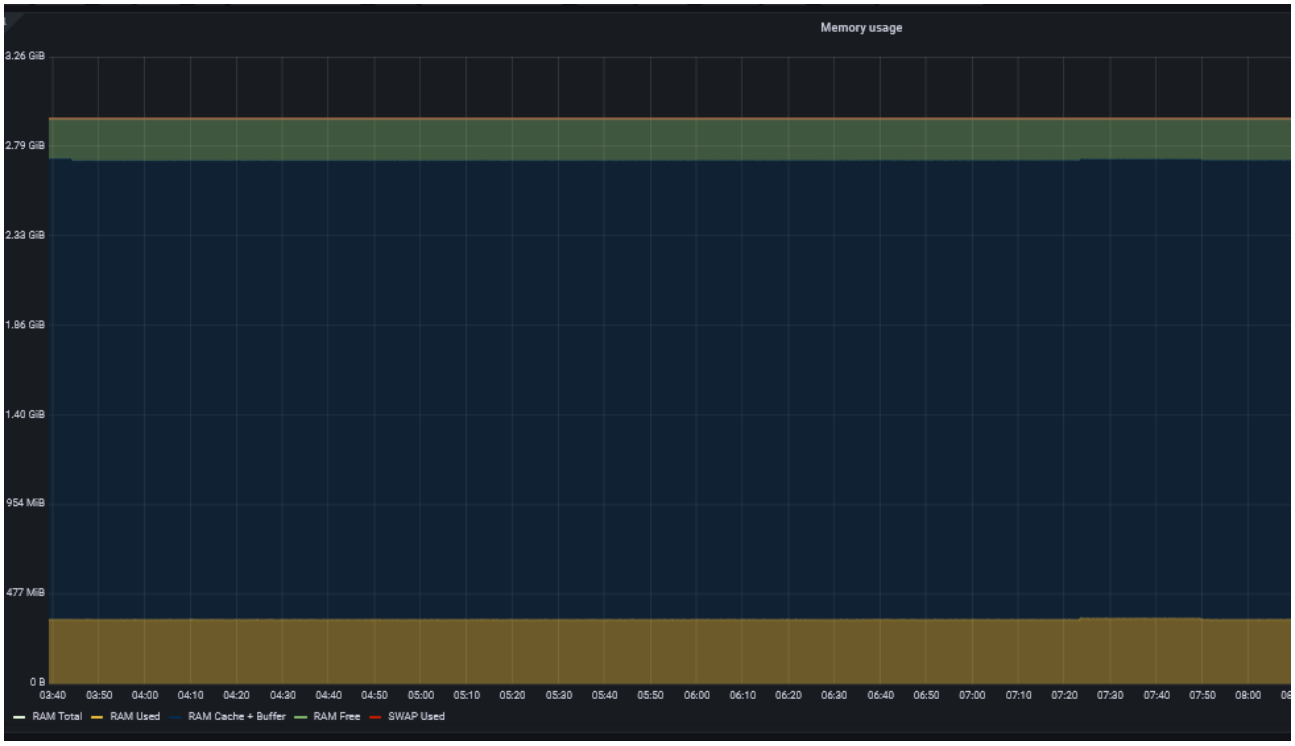

Figure 3.53: verifying metric collection

First, Figure 3.55 confirms that the cron-job we wrote is working as it should. Secondly, it confirms that retention condition is being applied and data is not exceeding 5GB. We already included this condition in Prometheus unit file.

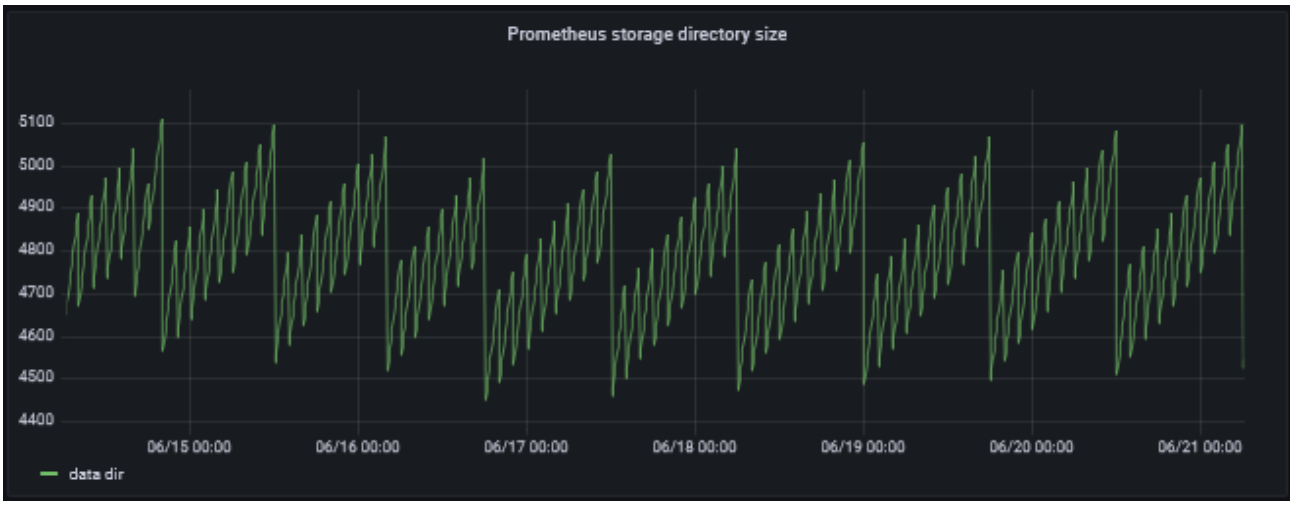

Figure 3.54: Prometheus data directory with retention applied

Condition in Unit file:

```
--storage.tsdb.retention.size=5GB
```

Figure 3.55: Unit File retention condition

3.11 Testing Alerts

This an example of how our alert is successfully triggered whenever a Promtail service is down, the alert provides a history of all previous states in the dashboard :

Figure 3.56: Firing Alert fro down instance

Figure 3.57 shows how a triggered alert for a java alert looks in the dashboard history:

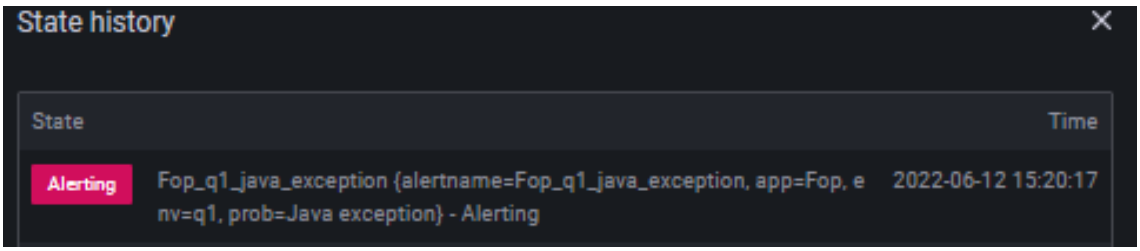

Figure 3.57: Firing Alert for java exception

3.12 Conclusion

In this chapter of our report, we have presented the realization of different aspects of our proposed solution. We started by metric collection, followed by querying and adding graphs and finally adding alerts and automating the installation process.

Conclusion et perspectives

General Conclusion

As a part of our graduation project, which was carried out within the company Sofrecom Tunisia, we designed and implemented a centralized monitoring and log management solution in order to add more visibility and observability to the VIO project.
In a first part, we presented the general context of our project. We introduced key concepts used in this solution followed by a comparative analysis of tools used in monitoring and log management. In a second part, we described requirement analysis and solution design. we focused on the identification of the actors, the analysis of the needs, and their communication between different components. Finally, we detailed the steps for the implementation of the solution and presented the obtained result.
This end-of-study project was an opportunity to introduce the concept of monitoring to all members of VIO Team and to adapt new technologies to develop the project.
This end-of-study project was a chance to acquire new skills in monitoring and log management. It was also beneficial from a personal standpoint, as it allowed me to understand the work process in a large company and in a professional hierarchy.
Perspectives
Even that the objectives of our project are accomplished , there's still some work to do.
First, we have to add a TLS configuration in the node exporters so that they be can securely scraped by Prometheus.
Secondly, we need to accomplish alert notification process and tune it.In fact, due to access problems to the Enterprise 's SMTP server sending mail notification wasn't fully established.

Bibliography

[1] *Sofrecom*. https://www.sofrecom.com/a-propos-de-nous/notre-entreprise.html/. Last access date : February 2022.

[2] *Agent vs. Agentless Monitoring*. https://sciencelogic.com/blog/agent-vs-agentless-monitoring/. Last access date : May 2022.

[3] *SNMP protocol*. https://www.ibm.com/docs/fr/spectrum-control/5.3.7?topic=standards-simple-network-management-protocol. Last access date : March 2022.

[4] *ICMP protocol*. https://www.fortinet.com/resources/cyberglossary/internet-control-message-protocol-icmp. Last access date : March 2022.

[5] *Effective Alerting in Practice*. https://newrelic.com/resources/ebooks/effective-alerting-guide/. Last access date : May 2022.

[6] *Graphite documentation*. https://graphite.readthedocs.io/en/latest/. Last access date : March 2022.

[7] *Zabbix documentation*. https://www.zabbix.com/manuals. Last access date : March 2022.

[8] *Graphite vs Prometheus*. https://metricfire.com/blog/graphite-vs-prometheus/. Last access date : March 2022.

[9] *Prometheus vs Zabbix*. https://techrepublic.com/article/prometheus-vs-zabix/. Last access date : March 2022.

[10] Brian Brazil. *Prometheus Up Running Infrastructure and Application Performance Monitoring*.

[11] Pedro Ara√ jo. *Hands-On Infrastructure Monitoring with Prometheus*.

[12] *Prometheus Documentation*. https://prometheus.io/docs/introduction/overview/. Last access date : February 2022.

[13] *Graylog documentation*. https://docs.graylog.org/docs. Last access date : March 2022.

[14] *ELK documentation*. https://www.elastic.co/guide/index.htmlz. Last access date : March 2022.

[15] *Loki Documentation*. https://grafana.com/docs/loki/latest/. Last access date : February 2022.

[16] *kibana documentation*. https://www.elastic.co/guide/en/kibana/current/index.html. Last access date : March 2022.

[17] *Grafana vs. Kibana*. https://betterstack.com/community/comparisons/grafana-vs-kibana. Last access date : March 2022.

[18] *Ansible documentation*. https://docs.ansible.com/. Last access date : April 2022.

[19] *Jinja2 templates documentation*. https://docs.ansible.com/ansible/latest/user_guide/playbooks_templating.html. Last access date : February 2022.

Annexes

Ansibles playbook

Below is a prometheus playbook in which we used variables through Jinja2 templates:

```
- name: ==> Configuration prometheus
  hosts: VM
  tasks:
     - name: Add prometheus user
       become: yes
       tags: useradd
       shell: "useradd --system --no-create-home --shell /usr/sbin/nologin prometheus"
       args:
         chdir: "/"
         warn: false
       register: useradd
       ignore_errors: yes
     - debug: msg="{{ useradd.stdout }}"
     - debug: msg="{{ useradd.stderr }}"

     - name: Déplacement du fichier prometheus
       become: yes
       tags: depzipprometheus
       shell: "cp -rfa /var/opt/data/flat/vioapp/prometheus-2.33.3.linux-amd64.tar.gz ."
       args:
         chdir: "/"
         warn: false
       register: depzipprometheus
       ignore_errors: yes
     - debug: msg="{{ depzipprometheus.stdout }}"
     - debug: msg="{{ depzipprometheus.stderr }}"

     - name: Extraction de prometheus-2.33.3.linux-amd64.tar.gz
       become: yes
       tags: extprometheus
       shell: "tar xvzf prometheus-2.33.3.linux-amd64.tar.gz"
       args:
         chdir: "/"
```

```
    warn: false
  register: extprometheus
  ignore_errors: yes
- debug: msg="{{ extprometheus.stdout }}"
- debug: msg="{{ extprometheus.stderr }}"

- name: creation de prometheus directory
  become: yes
  tags: dir
  shell: "mkdir /opt/prometheus"
  args:
    chdir: "/"
    warn: false
  register: dir
  ignore_errors: yes
- debug: msg="{{ dir.stdout }}"
- debug: msg="{{ dir.stderr }}"

- name: Déplacement de prometheus sous /opt/prometheus/
  become: yes
  tags: dep
  shell: " mv -v prometheus-2.33.3.linux-amd64/* /opt/prometheus/"
  args:
    chdir: "/"
    warn: false
  register: dep
  ignore_errors: yes
- debug: msg="{{ dep.stdout }}"
- debug: msg="{{ dep.stderr }}"

- name: Fix permissions
  become: yes
  tags: permission
  shell: "chmod -Rfv 0755 /opt/prometheus"
  args:


    -
    chdir: "/"
    warn: false
  register: permission
  ignore_errors: yes
- debug: msg="{{ permission.stdout }}"
- debug: msg="{{ permission.stderr }}"

- name: creation config file
  become: yes
  template:
    src: /home/ansible/delivery/ansible/playbooks/temp_prometheus.j2
    dest: /opt/prometheus/prometheus.yml

- name: creation de data directory
  become: yes
  tags: data_dir
  shell: "mkdir -v /opt/prometheus/data"
  args:
    chdir: "/"
    warn: false
  register: data_dir
  ignore_errors: yes
- debug: msg="{{ data_dir.stdout }}"
- debug: msg="{{ data_dir.stderr }}"

- name: Fix permissions
  become: yes
  tags: data_permission
  shell: "chown -Rfv prometheus:prometheus /opt/prometheus/data"
  args:
    chdir: "/"
    warn: false
  register: data_permission
  ignore_errors: yes
```

```
- debug: msg="{{ data_permission.stdout }}"
- debug: msg="{{ data_permission.stderr }}"

- name: Création d'un service systemd
  become: yes
  tags: creatserviceprom
  shell: "cp -rfa /var/opt/data/flat/vioapp/prometheus.service /etc/systemd/system/"
  args:
    chdir: "/"
    warn: false
  register: creatserviceprom
- debug: msg="{{ creatserviceprom.stdout }}"
- debug: msg="{{ creatserviceprom.stderr }}"

- name: systemctl daemon-reload
  become: yes
  tags: systemctlDaemonReload
  shell: "systemctl daemon-reload"
  args:
    chdir: "/"
    warn: false
  register: systemctlDaemonReload
- debug: msg="{{ systemctlDaemonReload.stdout }}"
- debug: msg="{{ systemctlDaemonReload.stderr }}"

- name: systemctl start prometheus.service
  become: yes
  tags: systemctlStart
  shell: "systemctl start prometheus.service"
  args:
    chdir: "/"
    warn: false
  register: systemctlStart


  - debug: msg="{{ systemctlStart.stderr }}"

  - name: systemctl enable prometheus.service
    become: yes
    tags: systemctlEnable
    shell: "systemctl enable prometheus.service"
    args:
      chdir: "/"
      warn: false
    register: systemctlEnable
  - debug: msg="{{ systemctlEnable.stdout }}"
  - debug: msg="{{ systemctlEnable.stderr }}"

  - name: systemctl status prometheus.service
    become: yes
    tags: systemctlStatus
    shell: "systemctl status prometheus.service"
    args:
      chdir: "/"
      warn: false
    register: systemctlStatus
  - debug: msg="{{ systemctlStatus.stdout }}"
  - debug: msg="{{ systemctlStatus.stderr }}"
```

Gitlab-CI file

Below is the file used to launch a pipline in Gitlab-CI through ODE:

```yaml
variables:
  # Indiquez ici la version du CLI à utiliser pour le déploiement.
  # L'équipe ODE communique à chaque nouvelle version, pensez à mettre à jour vos pipelines régulièrement.
  ODE_CLI_VERSION: "2.1.12"

stages:
  - install_prometheus
  - install_grafana
  - install_loki
  - node_exporter
  - promtail

Install_prometheus:
  image: dockerfactory.tech.orange/ode:$ODE_CLI_VERSION
  stage: install_prometheus
  before_script:
    - ''
  script:
    - ode add-configuration -n config_1 -i ansible/inventory ansible/playbooks/prometheus.yml
    - ode add-artifact -t config_1 ansible/playbooks/*.j2
    - ode deploy -t $ODE_ORANGE_CARTO_FOP_ID -A 2.9 config_1
  when: manual

Install_grafana:
  image: dockerfactory.tech.orange/ode:$ODE_CLI_VERSION
  stage: install_grafana
  before_script:
      ''
    -
  script:
    - ode add-configuration -n config_1 -i ansible/inventory ansible/playbooks/grafana.yml
    - ode add-artifact -t config_1 ansible/playbooks/*.j2
    - ode deploy -t $ODE_ORANGE_CARTO_FOP_ID -A 2.9 config_1
  when: manual

Install_loki:
  image: dockerfactory.tech.orange/ode:$ODE_CLI_VERSION
  stage: install_loki
  before_script:
    - ''
  script:
    - ode add-configuration -n config_1 -i ansible/inventory ansible/playbooks/loki.yml
    - ode add-artifact -t config_1 ansible/playbooks/*.j2
    - ode deploy -t $ODE_ORANGE_CARTO_FOP_ID -A 2.9 config_1
  when: manual

Node_exporter:
  image: dockerfactory.tech.orange/ode:$ODE_CLI_VERSION
  stage: node_exporter
  before_script:
    - ''
  script:
    - ode add-configuration -n config_1 -i ansible/inventory ansible/playbooks/node_exporter.yml
    - ode add-artifact -t config_1 ansible/playbooks/*.j2
    - ode deploy -t $ODE_ORANGE_CARTO_FOP_ID -A 2.9 config_1
  when: manual

Promtail:
  image: dockerfactory.tech.orange/ode:$ODE_CLI_VERSION
  stage: promtail
  before_script:
    - ''
  script:
    - ode add-configuration -n config_1 -i ansible/inventory ansible/playbooks/promtail.yml
    - ode add-artifact -t config_1 ansible/playbooks/*.j2
    - ode deploy -t $ODE_ORANGE_CARTO_FOP_ID -A 2.9 config_1
  when: manual
```