NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

DIOGO RODRIGUES MONTEIRO DA SILVA CAPITÃO

BSc in Computer Science

MONITORING IN CLOUD AND EDGE SYSTEMS

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon

April, 2024

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

MONITORING IN CLOUD AND EDGE SYSTEMS

DIOGO RODRIGUES MONTEIRO DA SILVA CAPITÃO

BSc in Computer Science

Adviser: Vítor Manuel Alves Duarte
Assistant Professor, NOVA University Lisbon

Examination Committee

Chair: Fernando Pedro Reino da Silva Birra
Full Professor, Faculdade de Ciências e Tecnologia, NOVA FCT

Rapporteur: Cláudio Miguel Sapateiro
Adjunct Professor, Escola Superior de Tecnologia de Setúbal do Instituto Politécnico de Setúbal

Adviser: Vítor Manuel Alves Duarte
Assistant Professor, Faculdade de Ciências e Tecnologia, NOVA FCT

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
April, 2024

Monitoring in Cloud and Edge Systems

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v6.10.17) [9].

Acknowledgements

Firstly, I want to start by thanking Professor Vitor Duarte, my advisor, for the support and guidance during the development of this dissertation. I would also like to thank NOVA School of Science and Technology for providing the resources and knowledge throughout the course. Finally, a special remark to my family and friends for the assistance and countless contributions that allowed me to finish my educational journey.

Abstract

Distributed systems monitoring is critical to ensuring the reliability, availability, and performance of complex systems; some of the main features are the early identification and resolution of issues before they turn critical, pinpointing possible bottlenecks, and possible optimizations in terms of performance and scalability. These aspects are especially prominent in Cloud infrastructures or Edge-based systems.

Due to the growth in the number of edge devices, new problems are emerging. The number of monitored targets has increased exponentially across different regions of the globe, creating an overload of data sent to a central node, thus, leading to bandwidth bottlenecking and an increase in latency. There is also an excess of information and metadata in the collected monitored data, which leads to issues when browsing through said data.

The main goal is to reduce the amount of data traffic and bandwidth usage when monitoring those systems while not only having information in the metrics collected about where said metrics are coming from and what services they are associated with but also allowing for low-latency alerts to be sent from an edge level.

With this solution, it is expected that the collected metrics will be more economical, organized in terms of size, and provide additional critical information, thus reducing bandwidth usage and better analysis of said metrics.

Keywords: Monitoring, Cloud, Edge, Fog, Prometheus, VictoriaMetrics

Resumo

A monitorização de sistemas distribuídos é crítica para garantir confiabilidade, disponibilidade e performance em sistemas complexos; algumas das principais características são a identificação e resolução atempada de erros antes que escalem negativamente, apontando possíveis *bottlenecks*, otimizações de performance e escalabilidade. Estes aspectos são especialmente relevantes em infraestruturas *Cloud* ou sistemas baseados em *Edge*.

Devido ao crescimento de dispositivos *Edge*, novos problemas emergem. O número de dispositivos monitorizados aumentou exponencialmente nas várias regiões do mundo, criando uma sobrecarga de dados enviados para um nó central, o que provoca *bottlenecks* de *bandwidth* e aumento da latência. Existe também um excesso de metadados e informação nos dados coletados, o que causa problemas ao navegar pelos mesmos.

O objetivo principal é reduzir a quantidade de tráfego e uso de *bandwidth* ao monitorizar esses sistemas, ao mesmo tempo em que se tem informações nas métricas coletadas sobre a origem das mesmas e a quais serviços ou aplicações a que estão associadas, além de permitir que alertas sejam enviados com baixa latência a partir de um nível de *Edge* ao invés de ter de chegar ao nó central.

Com esta solução, espera-se que as métricas coletadas sejam mais económicas e organizadas em termos de quantidade e que as mesmas forneçam informações adicionais, reduzindo o uso de *bandwidth*, levando uma melhor análise das mesmas.

Palavras-chave: Monitorização, *Cloud*, *Edge*, *Fog*, *Prometheus*, *VictoriaMetrics*

Contents

List of Figures

1

Introduction

The growth of cloud systems and edge devices has been rapid in recent years. Cloud systems, which allow for the storage and processing of data and applications remotely, have become increasingly popular as a way for businesses and individuals to access the resources and capabilities they need without having to invest in and maintain their infrastructure. With the expansion of these systems comes the widespread adoption of distributed applications at a global level. This architecture extends into the edge device level, providing points of presence closer to millions of users and improving availability and reducing latency. However, this also increases the complexity associated with monitoring, the vast network of edge devices and cloud systems requires sophisticated monitoring tools and techniques to ensure their seamless operation, as well as to minimize downtime and improve user experience. This complexity is further compounded by the increasing number of devices and applications connected, which leads to a need for efficient and effective monitoring solutions. Despite these challenges, the benefits of cloud and edge computing are too significant to ignore, making it a critical component of modern IT infrastructure and a driving force in the digital transformation of organizations worldwide.

Small edge devices are low-power devices placed at the "Edge"of a network, closer to the data source. They have also seen significant growth as they allow for the collection and analysis of data in real time without having to send it to a centralized location for processing. Together, these technologies enable new possibilities in areas such as IoT, Industry 4.0, and AI and drive innovation and efficiency in various industries.

With developments in these fields, a robust monitoring system must be put in place to ensure that they perform optimally and securely and detect and respond to any issues or failures. This is crucial for maintaining the availability and reliability of the services provided by these systems, as well as for ensuring the security of

the stored and transmitted data. Additionally, monitoring can also help to identify and troubleshoot performance issues, as well as to optimize the use of resources.

But monitoring such systems carries additional difficulties as there are an exponentially large number of devices to collect metrics from. Moreover, since most architectures focus on the flow of metrics into a single node, this will cause a bottleneck in the network bandwidth and latency.

This dissertation aims to create an approach that will avoid centralizing such amounts of data while adding additional features to optimize monitoring. The following chapters will provide further details on achieving the goal.

1.1 Problem

The research problem addressed in this dissertation is to understand the challenges and opportunities associated with monitoring cloud and edge systems. An exponential growth in these systems across several locations of the globe led to a whole new scale of monitored targets, which in turn implied vast amounts of data to be transmitted to a central node, including both structured and unstructured data.

With this widening flow of metrics into centralized nodes, scalability issues have also been presented; not only has the latency of data being sent increased, but bandwidth usage has also seen a steep escalation. Additional problems also need to be addressed in terms of the information and metadata collected from those targets as part of it is neglectable and, when not adequately filtered, causes not only a surge in the amount of data being sent to the central node but also a struggle when navigating through the data to observe the intended metrics. On the other hand, a lack of relevant labels can also factor in as a problem; the need for bottlenecking and service malfunction pinpointing creates the need for associating scrapped metrics with not only a geographical location but also with the service the edge device was accessing.

Finally, alert-sending problems arise as less bandwidth is available, and minor alerts take more time to reach the system's central node. As a result, they might have escalated into a critical situation by the time they reach the administrator. As latency increases and with the huge amount of metrics collected, alert-detection issues emerge, causing delays in the system's central node. Consequently, by the time alert notifications reach the system administrator, these alerts may have already escalated into critical situations.

Finally, the resources used by the monitoring system can compete with the

monitored system, introducing degradation in the target application/system thus incurring in extra costs.

1.2 Objectives

With the growing number of connected devices and the increasing complexity of data transmission and processing, it has become imperative to monitor and manage the performance and behaviour of edge devices. This dissertation aims at addressing significant challenges associated with edge and cloud monitoring and improve the overall system efficiency and performance by extending commonly used monitoring systems present in cloud and cluster environments such as Prometheus or VictoriaMetrics in order to better accommodate edge monitoring.

One of the primary goals of this dissertation is to reduce the amount of traffic flowing from the edge devices to the central node. This is relevant due to the fact theat large amount of data transmitted from edge devices can quickly consume available bandwidth and put a strain on the network performance. Reducing the amount of data transmitted can ensure that the network runs smoothly and efficiently.

In addition to reducing the amount of data transmitted, the dissertation also focuses on implementing additional labels that allow us to pinpoint the source of specific metrics and associate them with the corresponding service. This is important because it enables us to easily identify the source of issues and resolve them quickly and efficiently.

Moreover, the dissertation aims to ensure that critical alerts can be sent straight from the edge device level instead of having to wait for them to reach the central node. This is important because it enables us to respond quickly to critical situations and minimize the impact of potential issues.

Finally, it is important to note that a better monitoring architecture is essential to monitor and manage edge devices effectively. This involves designing a monitoring system that can effectively handle large amounts of data, quickly respond to critical situations, and provide actionable insights into the performance and behaviour of edge devices.

1.3 Approach

The dissertation plan to achieve the previously set goals of reducing the amount of traffic from edge devices to the central node, implementing additional labels,

and sending critical alerts from the edge level requires a comprehensive approach.

The first step is to choose the most suitable monitoring tool or even a combination of them, and then a flexible architecture for the cloud and edge environments by identifying their strengths and shortcomings. The next chapter will present a pool of monitoring tools to choose from. The choice will depend on several factors, such as the system requirements, the complexity of the network, and the desired level of monitoring.

Once the monitoring tool and architecture have been selected, the next step is to customize the most relevant metrics to be collected. This involves identifying the key performance indicators (KPIs) critical to the edge devices' performance and ensuring that the chosen monitoring tool can collect and analyze these metrics.

In addition, the plan also involves the implementation of additional labels that allow us to associate specific metrics with the corresponding service and location. This will enable us to identify the source of errors and resolve them quickly and effectively.

Finally, the plan includes implementing an alert system that can send critical alerts from the edge level. This involves adapting existing features or designing a new alert system that can quickly and effectively respond to urgent situations and provide real-time notifications to the relevant personnel.

2

Related Work

2.1 Monitoring options

This section starts with a brief introduction to what metrics are and why they play a significant role in monitoring systems, we will also be taking a look into the most common monitoring systems, their pros and cons, and how they collect and store metrics, with a brief overview of their architecture, and finally a selection of the best system/platform bearing in mind all the aspects previously stated and the requirements for the ideal solution for our distributed environment of cloud and edge infrastructure.

It should also be taken into consideration collected data from IoT environments, such as sensors. These application metrics could also be gathered using similar monitoring infrastructure.

2.1.1 What are metrics?

When regarding monitoring systems, metrics represent a quantitive measure of aspects of a system's performance, resources used and behaviour, their main purpose is to evaluate said aspects and allow for the systems manager to make well-informed decisions about its operation and maintenance.

This dissertation will focus on monitoring the following metric types:

- **Resource** - this type of metric revolves around the utilization of the system, for instance, CPU, memory, disk, and network usage. They provide insights into the performance and availability of the systems resources and help identify bottlenecks or inefficiencies.
- **Performance** - unlike the previous type, performance metrics' center of attention is around the system's execution and its components, such as

response time, throughput, and error rate. Together these metrics contribute to helping determine the overall effectiveness of the system as well as to identify areas of improvement.

To sum up, metrics represent the building blocks of thriving systems and are indispensable tools in the everlasting journey to maintain, manage and improve performance and adaptation to faults, load changes, and final user usage.

2.1.2 Prometheus

Prometheus[11] is an open-source systems monitoring and alerting platform. It was first developed at SoundCloud[1] and has since become one of the most popular monitoring tools for cloud-native infrastructure, it provides a flexible and scalable way to monitor the health and performance of applications, servers, and other systems by scrapping metrics.

Some of the features presented by Prometheus to stand out from the rest of the competition:

- **Data Model**, Prometheus chose to store data as time series, a series of data points belonging to the same metric associated with timestamped values.
- **PromQL**, a provided query language that will allow the user to select the desired data from a time series, aggregate it and display the result in a graph or as tabular data.
- Monitoring targets are discovered through static configuration or via service discovery, a component of its architecture. These targets can take the shape of actual infrastructure, an Operating System, or services and applications.
- **Pull System**, unlike other monitoring systems, Prometheus mainly relies on a pull system in order to collect metrics data, hence the term scrapping, this way it allows not only multiple instances to pull metrics instead of the services having to constantly push their metrics into the Prometheus Server making the actual monitoring tool the bottleneck of the service/application but also for improved insight on whether the service/application is still running.

From the figure 2.1, we can get a sense of this system's architectural components:

[1] https://developers.soundcloud.com/blog/prometheus-monitoring-at-soundcloud

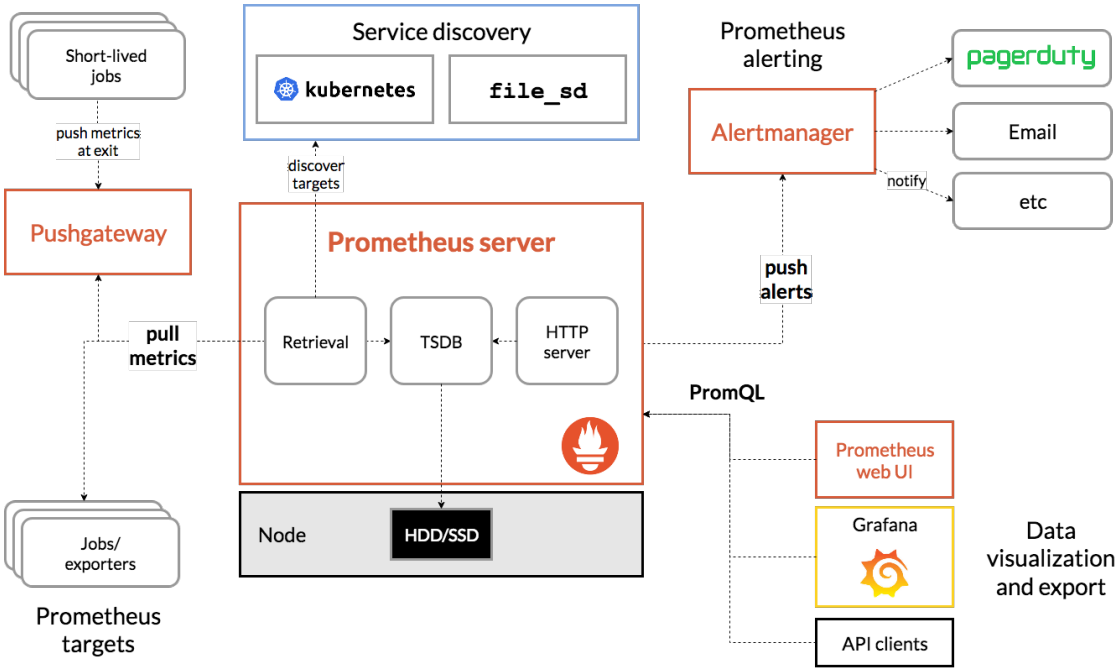


Figure 2.1: Overview of Prometheus Architecture

- **Prometheus Server**, responsible for the monitoring work, divided into three parts:
  - **Time Series Database** - responsible for storing the metrics data.
  - **Data Retrieval Worker** - has the sole purpose of pulling metrics data from its targets and, in turn, pushing them into the TSDB.
  - **Server API** - handling the queries and directing them to the TSDB.
- **PushGateway**, ideal for monitoring targets that only run for a short time, allows services that fit this description to push their metrics directly into the Prometheus TSDB, these components represent an alternative and shouldn't be the main focus of the use of this tool.
- **Alert Manager**, a component that has previously defined rules and recipients for when alerts should be triggered. These alerts can be notified through platforms such as email or Slack

Regarding scrapping data, some services already expose Prometheus endpoints, but the majority require other mechanisms.

The implementation of exporters can solve this; services that will scrape specific types of metrics from a designated target, convert them into a format Prometheus can understand, and ultimately expose them in an HTTP endpoint.

During research and practical experiments, the most frequent exporters used were:

- **Node Exporter** - Exposes hardware and OS level metrics designated to monitor the host system and is not recommended to be deployed in a containerized environment.
- **BlackBox Exporter** - Allows for the scrapping of metrics from HTTP, HTTPS, DNS, TCP, ICMP, and gRPC endpoints. This specific exporter uses a multi-target exporter pattern meaning that the metrics will be scraped via a network protocol, the exporter is not required to be running on the machine the metrics are being scrapped from, and the exporter can query different targets
- **Promxy**[7] - its main purpose is to aggregate Prometheus shards are then presented to the user as a unique API endpoint, simplifying operations and scaling when there is a single host server.

One of the key advantages of using Prometheus is its reliability. Although each Prometheus Server instance is standalone, as they do not depend on any network storage or other remote services, it has the ability to work even if other components of the infrastructure are not functional and is also straightforward to set up as it does not require extensive infrastructure in order to get it started.

Moreover, its open-source nature and active community of contributors means that new features and improvements are regularly added, making it a versatile and cutting-edge monitoring tool.

While Prometheus has many strengths as a monitoring system, it also has a few limitations that may make it less suitable for specific use cases. One of the main disadvantages of Prometheus is its lack of scalability. If you have multiple Prometheus Servers aggregating large amounts of data via functions such as sum, count, or average, it can become challenging to configure Prometheus to achieve optimal metric gathering and processing. This can be complex and force the user into evaluating trade-offs due to the fact that, on the other hand, is the option of having a unique Prometheus Server doing all the metric scrapping, but in doing so it will then limit the number of metrics that can be monitored.

Additionally, the memory usage of the database and metrics exporters can lead to memory issues when dealing with large amounts of data, especially if sharing resources with application components or in resource-constrained environments.

Finally, Prometheus does not have built-in security features, such as authentication and authorization, making it vulnerable to security threats. This can make it

challenging to secure Prometheus in highly regulated environments and to ensure the confidentiality of sensitive performance data.

To counter the scalability issues, Prometheus offers the **Federation** functionality, with the main goal of allowing a Prometheus Server to scrape metrics from other instances of Prometheus Servers. There are two types of federation:

- **Hierarchical federation** - allows for the scaling of systems that oversea millions of nodes. Its topology would then resemble a tree, with higher-tier instances scrapping metrics from lower-tier Prometheus instances.
- **Cross-service federation** - a Prometheus Server instance is configured to scrape specific metrics from other services in order to enable alerting and queries in both datasets with a sole representative.

While this sounds like a solution for the previously stated problems, several issues arise from using said configuration, such as scalability limitations, as this would generate significant overhead from aggregating large amounts of metrics from different sources resulting in slow query times and potential data loss, network latency issues as edge level devices are scattered around the globe causing slow data transmission making resource usage, like memory and CPU, in the central nodes and on the edge challenging to aggregate data in real-time, and complexity of management despite being simple to set up a standalone server, configuring several instances to communicate with each other can become a hassle.

2.1.3 VictoriaMetrics

VictoriaMetrics[18] is a fast, scalable, and efficient open-source monitoring solution and time series database designed to handle large amounts of time-series data. It was created to address the limitations of other existing monitoring solutions and to provide a high-performance alternative for organizations dealing with large amounts of data.

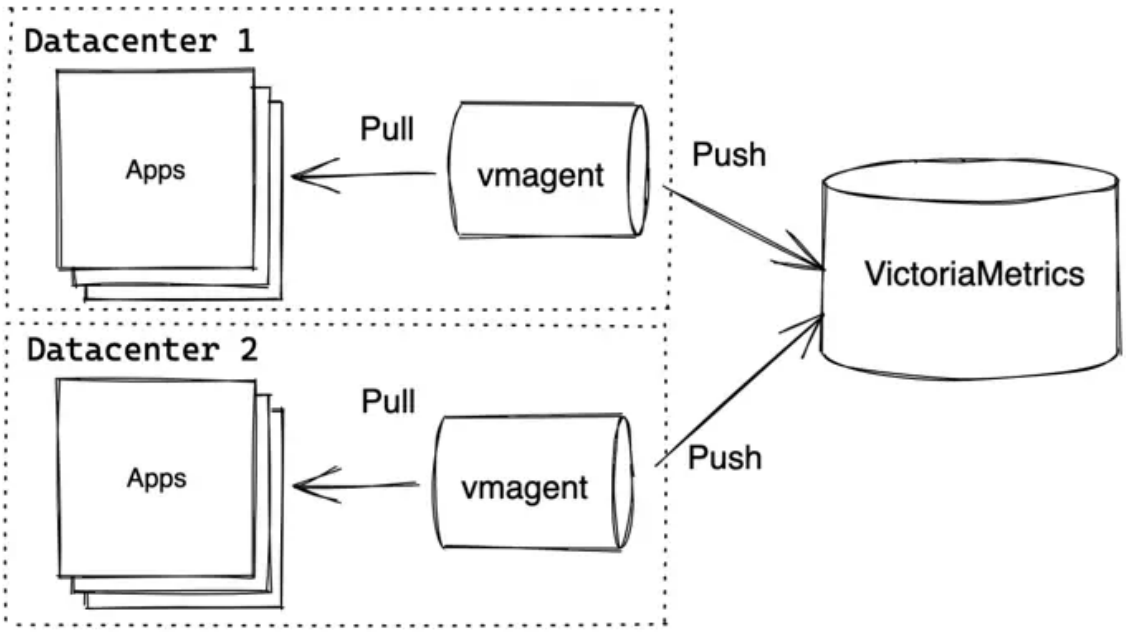


Figure 2.2: VictoriaMetrics architecture general concepts.

It is known for its ability to handle large amounts of data with high performance and low resource utilization. It can store, index, and query time-series data with low latency, making it a suitable solution for organizations dealing with large amounts of metrics data. VictoriaMetrics supports a variety of data sources, including Prometheus, OpenMetrics[2], and Graphite[3], and can easily integrate with other tools and systems in an organization's infrastructure. Additionally, VictoriaMetrics has a built-in alerting system that can be used to send notifications and take automated actions in response to changes or anomalies in the data.

With the goal of being able to deal with vast amounts of time-series data such as telemetry data from cloud-native applications, infrastructure, and also edge devices this system was designed and built around 3 core principles cost-efficiency, operational simplicity, and scalability.

Recommended also for organizations dealing with high-cardinality data, where there are many unique metrics and values, as it can handle said data with low latency and high performance. VictoriaMetrics is also commonly used for monitoring and alerting in large-scale cloud-based environments, such as Kubernetes[4] and OpenStack[5], where performance and scalability are critical. In terms of its components, they were designed to be independent whilst also being part of a wider system. At the center of its operations is the VictoriaMetrics TSDB which can exist

[2] https://github.com/OpenObservability/OpenMetrics
[3] https://graphiteapp.org/
[4] https://kubernetes.io/
[5] https://www.openstack.org/

in a single or cluster version, compatible with the Pull model from Prometheus with support for different types of data transfer methods and ways of querying data such as Grafana[4], PromQL, and Graphite. The single version has the ability to scale up to substantial values and like Prometheus supports both push and pull models, has integration with Grafana, and allows for target scrapping all in one node.

On the other hand, we have the VictoriaMetrics cluster which represents the scalable component, providing horizontal and vertical scaling of different types of loads and coming with sharding and replication.

In order to push and pull metrics from a discovery service it makes use of a lightweight and performant agent, **vmagent**, sending the metrics to a centralized storage, it also has the ability to temporarily store metrics in case the TSDB is not available, forwarding those metrics once the connection is re-established.

With the goal of sending alerts in case of malfunction or some pre-determined rules, VictoriaMetrics has another assigned component for this duty, **vmalert**, similarly to Prometheus it will first evaluate the rules and conditions with the special ability to allow to replay past rules.

Aiming to function incrementally and disregarding constantly repeated values in order to only upload new relevant events this system provides yet another agent, **vmanomaly**, that gives the ability through their machine learning software to visualize data and provide the user with options on whether they want to alert or visualize anomalies. Regarding security, all communications with VictoriaMetrics are possible via TLS with HTTPS, if needed, in a cluster environment there is also the possibility to implement a no-trust policy by using mTLS.

VictoriaMetrics also claims to have features that distinguish them from other monitoring tools with some of them being:

- **Smallest Disk Storage Size**[17]: Aliaksandr Valialkin was able to conclude that when put up against other Time Series Databases in a benchmarking suite, VictoriaMetrics results were significantly better than other supported systems such as Timescale[6] and InfluxDB[7], after ingesting 1 billion datapoints from 40 thousand different time-series, Timescale occupied 75 times more disk space than VictoriaMetrics and InfluxDB occupied a third more than VictoriaMetrics.

- **Highest Ingestion Rate**[15]: In the same suit VictoriaMetrics proved that

[6] https://www.timescale.com/
[7] https://www.influxdata.com/products/influxdb-overview/

with increasing CPU capabilities and RAM, its data ingestion rate far surpassed its competition by as much as 4 times in the top-tier setup.

- **Fastest Query Performance**[14]: After inserting 525.6 billion data points into a testing suite, VictoriaMetrics was able to query over all rows at a speed of more than 1 billion rows per second.
- **Lowest Memory Usage**[16]: When comparing it with other flagship monitoring solutions, in this case, Prometheus, VictoriaMetrics shows a consistent use of 5.3 times less RAM whereas Prometheus had spikes of up to 23GB of RAM usage.

On the other hand, one of the disadvantages of VictoriaMetrics is that it is relatively new compared to other monitoring systems and may not have the same level of community support or third-party integrations as established tools. Additionally, the user interface and query language in VictoriaMetrics may be more complex for some users compared to other monitoring systems, requiring a steeper learning curve for some organizations. Despite these limitations, VictoriaMetrics remains a popular and highly-regarded notation for organizations dealing with large amounts of time-series data.

2.1.4 OpenNMS

OpenNMS [13] is an open-source, enterprise-grade network monitoring platform that provides a comprehensive solution for managing and monitoring IT infrastructure. It was first developed in 1999 and has become one of the most popular network monitoring tools for organizations of all sizes.

This platform provides a wide range of functionalities for network monitoring, including asset management, performance monitoring, and fault management. It has a robust event management system, which allows administrators to receive real-time notifications of changes or issues within the network.

Additionally, OpenNMS provides a comprehensive reporting system that will enable administrators to extract insights and trends from the collected data, making it easier to identify potential problems and optimize performance.

As a result, it is well-suited for organizations with complex network infrastructures, such as large enterprises and service providers. It is also commonly used by organizations that need to monitor large numbers of devices, such as data centers and cloud environments. OpenNMS is particularly well-regarded for its ability to manage and monitor large, different networks, making it a popular choice for organizations dealing with diverse IT environments.

This system comprises four different components:

- **OpenNMS Horizon** - works by collecting data from network devices using a variety of protocols and presenting that data to users through a graphical user interface. To perform monitoring functions, it uses the following components:
  - **Core** - This is the central component of the OpenNMS Horizon system that provides the core functionality for managing and monitoring network devices. It includes the web-based user interface, the database, the alerting system, and other vital components.
  - **Minion** - This remote monitoring agent runs on network devices and performs data collection tasks. Minions can be deployed on network devices to monitor their performance and to provide data to the OpenNMS Horizon Core system. In addition, minions can be used to monitor devices in remote locations, allowing organizations to extend the reach of their monitoring systems.
  - **Sentinel** - This is a real-time event monitoring and alerting system that integrates with the OpenNMS Horizon Core system. Sentinel allows users to monitor events as they occur and to receive alerts in real time. In addition, sentinel provides a flexible and powerful event processing system that can be used to detect and respond to network and device issues quickly.

- **Helm** - which allows users to customize their dashboards using performance and fault management

- **ALEC** - its main purpose is to improve root-cause analysis by grouping related alarms that were likely to be generated by a similar issue. It starts by analyzing the provided network and then converting it into a graph, then some metadata is added to the alarm after this, these alarms are attached to the graph after being triggered and grouped by the root cause. When the number of related alarms passes a certain threshold, an event is then sent to OpenNMS.

- **PRIS** - used when discovery is not able to facilitate target uncovering when a large network is put in place, it remodels extracted data previous to start monitoring with OpenNMS.

To sum up, and after further analysis of the documentation, when monitoring networks OpenNMS is a strong candidate for a strong suited system incorporating several advantages such as:

- **Delegating scrape effort** - in order to better distribute the scrape load, the use of minions carries several resource improvements.
- **Manifold protocol integration** - allows for the integration of a wider range of devices and systems.
- **Large set of features** - the most notable being event management which allows for the network manager to receive real-time notifications, network discovery, and performance monitoring.

But on the other hand, it has its own drawbacks, some of them being:

- **Resource-intensive** - it requires large amounts of memory and processing power when monitoring the network causing increased latency and restrictions in bandwidth.
- **Steep learning curve** - Challenging to grasp especially for first-time users in network management and monitoring.
- **Potential for alert delays** - Although this is a challenge faced by several network monitoring systems, given a large number of devices, OpenNMS may experience some latency in alert reporting due to its event sending mechanism.

Despite showing promising features such as the use of minions to improve metric scraping , due to its scrape method of polling metrics, scalability will become an issue when faced with millions of devices to scrape metrics from. Additionally, lacks the ability of relabeling said metrics, restraining the system administrator from being able to differentiate their origin in terms of application, region or service.

2.1.5 M/Monit

M/Monit[10] is an open-source monitoring platform that extends on a small, popular open-source tool called Monit, used to administer and keep an eye on Unix systems, as a comprehensive solution for managing and monitoring IT infrastructure. It is designed to be simple yet powerful and flexible, making it a suitable solution for organizations of all sizes. It provides various functionalities,

including system and process monitoring, network and service monitoring, and log analysis. It also provides a centralized management console that allows administrators to easily manage and monitor multiple servers and devices from a single interface.

Additionally, M/Monit has a built-in alerting system that can be used to send notifications and take automated actions in response to changes or issues within the environment.

In order to provide the high performance the system uses pools of threads with an event-driven I/O architecture.

Its architecture relies mainly on the use of a monitoring program running on all hosts being monitored, **Monit**. Regularly, this program will send messages to the platform providing insights on the host's status, whether it is up and running or offline, and if there is a problem regarding that service a message will also be sent to the platform informing it of the incident. All these types of messages will be stored in a database; after these messages are received, M/Monit will check its set of rules, and if they coincide, then a notification will be sent to the system manager.

After the metrics are collected, the system administrator can check the analytics report for the values of those metrics and run queries. Additionally, there is a feature for trend prediction where the user can observe possible future values based on current metrics.

The main selling points of this monitoring system are:

- **Easy of use** - simple setup process with intuitive configuration without requiring components from third-party services/applications.
- **Variety of Monitoring options** - M/Monit has a wide variety of monitoring options ranging from servers, processes, cloud, disks, files or folders.
- **UI** - simple, intuitive, and scalable interface whether you are managing small or large amounts of hosts. This interface is also available in mobile format through the use of a browser.
- **Open Source** - there is a possibility of using source code with a complete build system.

Lacking fundamental monitoring aspects in order to become a viable solution, this systems fails to address its scalability solutions and explain how it could be reliable in a monitoring environment where millions of nodes from different regions are sending data to a central cloud node. In spite of retrieving relevant

metrics such as CPU and memory usage it also lacks an important feature, the use of additional labels for better group collected metrics.

2.2 Dealing with Cloud and Edge Environments

This section is reserved for addressing related work where the authors encounter similar issues regarding monitoring an Edge or Cloud context and the options they propose in order to deal with and mitigate them.

- **A Comparison of Monitoring Approaches for Virtualized Services at the Network Edge**[5]: Marcel Großmann and Christian Schenk, start this paper by introducing issues that come with the rapid growth in IoT previously mentioned, they present 2 approaches. The first one is the development of 5G which aims at increasing network capacity, lower latency, energy consumption, and operational costs. The second approach comes with the implementation of fog computing which would utilize the widespread presence of IoT devices and gateways to reduce network traffic by relocating computation closer to the network edge.

  However, the implementation of fog computing requires an investigation of **Quality of Service**, (QoS), drawbacks, and an increase in the performance of fog-enabled services. A challenge in achieving this goal is the need to keep track of available resources on a potentially large number of fog nodes with varying compute, network, and storage capabilities. There is a need for low overhead monitoring solutions for fog nodes, which are likely to be small and less powerful ARM-based Single Board Computers. After this, they introduce the paper's main goal which is to compare existing monitoring approaches in terms of their CPU and RAM utilization on low-power devices like the Raspberry Pi 3 and evaluate their applicability as the foundation for a fog computing framework.

  The next focus was on determining the CPU and RAM overhead added by the monitoring frameworks and to achieve this 2 frameworks were evaluated, PyMon and Prometheus, both were deployed on Raspberry Pi clusters and to obtain statistics they used an integrated tool in Docker.

  The following text presents some important remarks about the performance measurement results of the resource utilization of three monitoring applications: Monit, Node exporter and cAdvisor[3]. The tests were conducted on an RPi cluster with three devices and the Prometheus server and Grafana

were deployed on the same node. The measurements are divided into three parts based on the different processing steps of a monitoring framework: monitoring, aggregation, and visualization.

In terms of CPU Usage, results show that Node exporter utilized a substantially lower percentage than Monit, and if the polling interval decreased then the difference increased even more, scalability wise, Monit also has poor performance due to high implementation overhead and very few containers allowed to be monitored on a single RPi 3 while cAdvisor, another exporter associated with Prometheus, had low overhead and scaled well with more containers.

Another measurement mentioned was RAM Usage, in which PyMon had a surprisingly better performance since it allocated less RAM than Prometheus but with an update and with the use of metrics caching Prometheus is expected to overcome these results and prevail as the most adequate monitoring tool.

Furthermore, it was given another advantage to Prometheus due to its adaptability and compatibility with multiple different exporters to retrieve different types of metrics. Lastly, the representation of scraped metrics was given increased importance, here Grafana paired with Prometheus takes the win since it has several different customizable dashboards to display information.

The overall conclusion is that Prometheus along with its Exporters makes use of fewer resources and is the most adequate solution to monitor fog nodes without producing large amounts of overhead. Nonetheless, this overhead should not be overlooked due to the fact that it could become an issue when the system escalates.

- **FMonE: A Flexible Monitoring Solution at the Edge**[1]: This paper recognizes the same problems previously mentioned when discussing Cloud and Edge environments. To tackle these issues, it also considers the important concept of **fog computing**; this term refers to a new paradigm where the operation takes place between the cloud and the edge, helping to bridge the gap between these two environments.

  In the context of this paper, a proposition is made where a central node is still put in place, representing a cloud entity. Still, the difference to other implementations comes after, where some intermediary nodes are allocated before the edge-device level. This type of procedure allows for

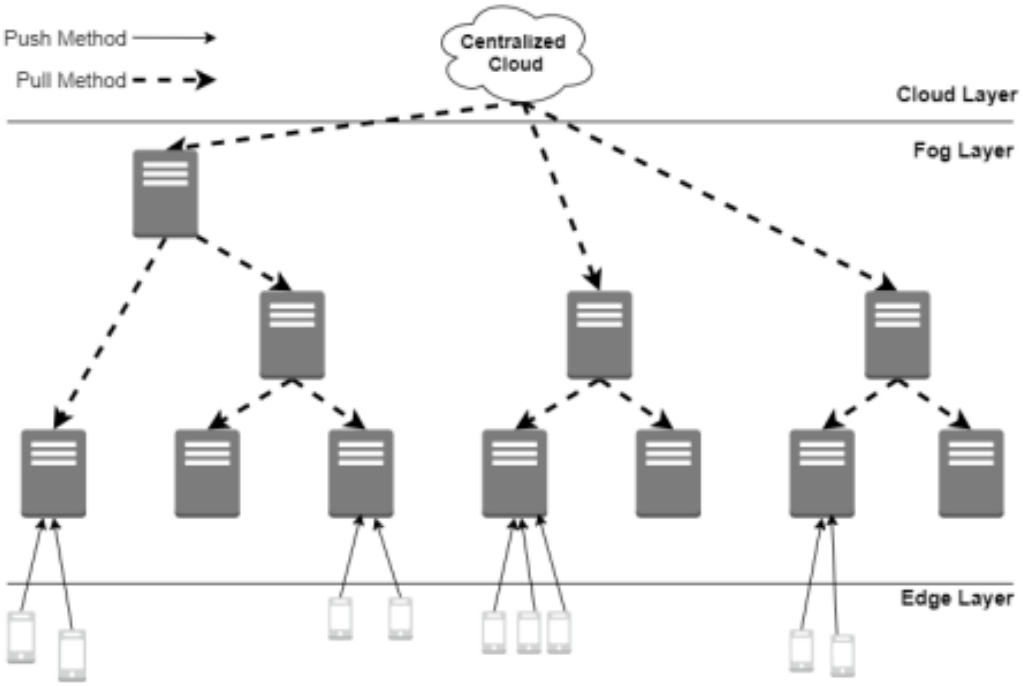


Figure 2.3: Exemplification of monitoring architecture with a fog layer

some degree of filtering and pre-processing of data fundamental in the current paradigm, where it would allow for a reduction of data sent through the network, thus saving bandwidth and a potential decrease in latency. Another proposition worth mentioning is the fact that this paper aims at conserving the collected metrics in a local database before sending the bulk of the data to the centralized cloud.

Despite tackling most of the relevant problems in their proposed solution, the latency for the metrics that reach the cloud will increase and they pretend address edge-level alert management as future work.

- **REDEMON**[2]: Presented as a monitoring system for edge devices, REDEMON aims at solving monitoring issues in an established network called Guifi.net[19], more precisely in Barcelona. They point out the lack of robustness and resiliency when dealing with increased scaling of edge devices in the current system and present REDEMON as its successor offering reduced resource consumption in terms of CPU, RAM, and bandwidth while maintaining the goal of limiting the number of computing resources used by the monitor agent.

  By creating a more decentralized and redundant monitoring solution, the authors were able to achieve the set goals; they maintained a low storage

consumption and improved CPU and memory consumption allowing for the devices to maintain a reasonable amount of computing resources to operate other services without compromising their primary purpose.

- **Optimization of Edge Resources for Deep Learning Application with Batch and Model Management**[8]: The goal of this article is to give more insights into how to optimize the deployment of AI on edge devices. Still, as expected, the authors ran into issues related to the limited amount these devices have to work with, as well as the challenges presented by scaling in this environment. One of the proposed options to combat this was the implementation, seen previously, of distributed workloads between a cloud and fog layer. Still, it was noticed that in doing so, the quality of service was affected negatively.

  In a later subsection, they refer to the monitoring of GPU resources in the edge devices hosting the AI model, and the tool chosen to achieve this was Prometheus since it gathered the flexibility requirements needed. In order to achieve the coveted ideal batch size to update the AI training model, they used Prometheus together with one of its exporters, Node Exporter, in order to monitor these devices' resource usage. Then they took the scarped metrics and inserted them into predetermined criteria.

- **Reviewing Cloud Monitoring: Towards Cloud Resource Profiling**[6]: The article reviews state-of-the-art monitoring approaches of cloud data centers. It discusses the growth of cloud data centers and the importance of cloud computing in data center operations, with Infrastructure as a Service as the most important model. Transparency in resource consumption is important for trust between customers and service providers. Both the cloud customer and the cloud service provider have a great interest in monitoring their physical and virtual infrastructures and current solutions consist of distributed components, such as adaptors or collectors, that collect metrics and transfer them to a central node. The paper aims at providing a novel approach for cloud monitoring by combining the customer and provider monitoring systems into one platform and reducing the monitoring overhead through statistical computations.

  The problems stated focus on the difficulties in collecting and working with resource statistics in cloud data centers. The first group of problems is related to collecting resource statistics. There is a lack of access to resource utilization metrics between physical and virtual levels due to isolation. Overbooking

and resource sharing leads to situations where the load experienced by a virtual machine is not accurately reflected on the physical level. As a result, it is important to identify resource bottlenecks for VMs on the physical level. Additionally, resource utilization metrics can vary between different hardware, making it difficult to compare data from different servers. Therefore, it is important to combine multiple metrics to understand the virtual resource experiences.

The second group of problems is related to handling resource statistics. Existing monitoring solutions generate a large amount of data, which requires significant resources for processing, transportation, and storage. Time series database systems like InfluxDB and Prometheus are used to manage this data, but the growth of collected statistics can be problematic. Additionally, when working with stored resource statistics, the data has to be read and analyzed, which can be time-consuming and resource-intensive. The question arises of why to store such a large amount of data when it is only used for resource scheduling, alerting systems, or graphical utilization overviews. The assumption is that time series can be computed to a statistical profile at its source, where the monitoring agents or probes are located.

The solution proposed revolves around the creation of Clomon, a novel approach to cloud monitoring for cloud data centers. It aims at saving resource capacities and supporting resource allocation, alerting, and visualization by its design. Clomon runs on each physical host and acts as a collector, but instead of transferring the resource utilization values, it converts the data to produce a statistical representation of the resource utilization. This statistical profile, which is updated continuously, contains all the necessary information for applications such as dashboards, data center optimization tools, and alerting engines. It has three main components: connectors, collectors, and the profiler. The collectors gather resource utilization metrics, while the profiler compiles the resource utilization profile for each workload item. The data collection addresses the challenges of overbooking, hardware independence, and application-specific metrics.

Finally, after identifying problems with current cloud monitoring solutions and presenting a new approach to meet the main motivations of cloud monitoring such as alerting, resource allocation, and visualization, instead of transmitting and storing raw time-series measurements, this new approach used statistical profiles to reduce computation, network, and storage requirements. The conversion of resource statistics into profiles requires

computation time on each host, but it is assumed that this can be done with simple statistical computations. In turn, this conversion could potentially be done in a fog layer where operations over collected data have shown improvements at the network level with a reduction in bandwidth use.

- **CloudProcMon: A Non-Intrusive Cloud Monitoring Framework**[12]: This paper mentions 2 different cloud monitoring types intrusive and non-intrusive. The first one involves the insertion of monitoring agents into the monitoring target, with drawbacks such as overhead on the targets and dependence on the availability and uptime of the target. Non-intrusive monitoring, on the other hand, does not require any monitoring agents, data is collected either through the monitoring host or by a cloud component.

  However, the slow polling frequency of cloud component data collection and the challenges posed by operating-system-based non-intrusive monitoring have been noted. The authors then propose their own performance monitoring framework, CloudProcMon, which is non-intrusive, lightweight, and scalable both horizontally and vertically and solves the challenges posed by other monitoring frameworks.

  Their implementation consists of monitoring targets running on a cloud compute node by exploiting the information from the host OS's virtual file system and collecting CPU usage, network utilization, and memory usage metrics. It can be run in two modes: the push protocol, where it sends the data directly to the monitoring dashboard on the cloud controller node, and the pull protocol, where it waits for a request from the monitoring dashboard.

  The authors then compared their implementation against a known cloud monitor, Nagios[8], more specifically the Nagios-Ceilometer Plug-in (NCP). CloudProcMon was found to have a quicker monitoring latency with a real-time of 3.45 sec compared to NCP's real-time of 11.22 sec. The polling frequency of this tool was also found to be more accurate as it detects changes in CPU load without causing any overhead. On the other hand, NCP was unable to detect changes in CPU load until 10 minutes and caused high monitoring overhead with a CPU load increase of 10-20% CloudProcMon only had a 2-4% difference in CPU load with and without monitoring, proving that it creates almost negligible monitoring overhead.

  In spite of their results showing evidence that non-intrusive cloud monitoring is more efficient and has better scalability for performing cloud monitoring

[8] https://www.nagios.org/

that creates negligible overhead, their experiments only took into consideration a small pool of metrics, meaning that shortcomings are expected when the amount of devices and metrics grows exponentially.

2.3 Relabeling

Aiming at solving sharding problems when collecting several metrics from different geographical spaces, both Prometheus and VictoriaMetrics grant the ability to relabel collected metrics.

By definition, relabeling refers to the process of modifying or transforming metric labels within a monitoring system. This can be used to filter, aggregate, or re-map metrics based on specific criteria, such as grouping metrics by application, service, or environment or removing unwanted metrics based on their label values.

The implementation of such a technique in the proposed system would allow for aggregation in terms of the source of metrics, meaning that instead of receiving millions of metrics from different targets, we would be able to aggregate metrics by region as well as by service, this way the user would be able to discriminate the data scrapped and determine where the focus of a particular problem lies as well as it can help identify patterns and trends that might not be visible when looking at the data in isolation. Relabeling metrics can also help improve the accuracy and efficiency of alert management systems. By grouping related data together, it becomes easier to set thresholds and triggers that are relevant to specific regions or services. Another advantage of performing this optimization would be the possibility of duplicate filtering metrics, thus, reducing the number of metrics scrapped, leading to even more reduction in network bandwidth usage and better latency overall.

2.4 Alert Managing

In terms of alert managing, given the fact that the bandwidth can be minimized by reducing the scrape frequency and the volume of data collected (filtering and aggregation of metrics), some alerts can be detected later or even be missed. This alert triggering might be pushed onto lower layers such as fog or even edge to allow for detailed detection and shorter latency notification.

Alerts would then be propagated faster, causing improved detection of malfunctions as well as the ability to pinpoint what the cause of the error may be and whether the error is happening in a specific region or due to an issue with

one particular service, thus, improving productivity and preventing minor errors from escalating to the rest of the system.

Additionally, an improved alert management system can also help to reduce false positives and false negatives. A false positive occurs when an alert is triggered unnecessarily, while a false negative occurs when an alert is not triggered even though there is a problem. Both of these scenarios can lead to significant costs, including wasted time and resources, as well as damage to the devices or data. A well-designed alert management system helps to minimize these types of errors, ensuring that administrators receive accurate and relevant information about the status of edge devices and services.

3

Architecture proposals

In this chapter, will go over various architecture solution options along with the various components and how they relate to one another and the levels in which they are located aiming at providing example scenarios that show layouts with different component and processing distributions throughout the infrastructure, from the cloud to the edge layer.

Given the initial objectives and the conclusions gathered from the previous chapter, we are able to proceed to implement cloud and edge based system configurations that include Prometheus or VictoriaMetrics as the monitoring system. Not only do they show great potential in terms of scalability but also through the use of intermediary nodes to further reduce the amount of data sent to the central node.

3.1 Main components

The following proposals are based on Prometheus, VictoriaMetrics or a mixture of both, the architectural designs aims at comparing these systems and to address the challenges inherent when monitoring complex and large scale infrastructures. These proposals display previously discussed scenarios: cloud based where all processing is taken care of by the cloud node and, as mentioned in FMonE framework, a solution based on delegation processing tasks such as metric aggregation and filtering to fog/edge layer nodes in order to distribute the load and mitigate network traffic that would otherwise be addressed at the cloud layer level. To further reduce network traffic measures were implemented such as reducing the scrape interval of scraping agents. Finally to improve alert detection, edge layer alert components were deployed in order to promptly evaluate given rules.

The main components used in these system configurations are:

- **Prometheus Server** - responsible for scraping and storing metrics from various data sources with these being applications, servers or network devices and evaluate rules that can lead to triggering alerts when specific metric values exceed predefined thresholds.

- **VictoriaMetrics Single** - Similar to the Prometheus server, this component will scrape metrics from different data sources and store them for further evaluation and possible alert triggering. It will also take advantage from its micro service driven architecture and various agents in order to optimize the usage of previously mentioned resources such as *vmagent* and *vmalert*. In this cases its sole purpose would be to store metrics for the configured time interval.

- **Alert Manager** - Configured to orchestrate the propagation of alerts triggered by the previously mentioned storage options. In case of Prometheus as the data source the alerts are sent directly to the Alert Manager. On the other hand, with VictoriaMetrics Single the alerts will be sent by the **vmalert** component.

- **Grafana** - The main reason for the use of Grafana lies in its ability to convert scraped metrics into visually engaging dashboards, graphs, and charts. It is an excellent tool for presenting information in an understandable format, whether it is used to render time-series data to track performance trends over time. In this context we will take advantage of said features aiming at better analyzing the received metrics in a more detailed way while also portraying the benefits of relabeling these metrics.

- **Node Exporter** - As a metrics collector for machine resources it provides detailed information about the functionality and state of the underlying systems. The Node Exporter interfaces directly with the kernel of the operating system to collect a wide range of metrics, including CPU, memory, disk I/O, network activity, and other critical system parameters. In the context of this dissertation the Node Exporters are deployed to personify monitoring targets and to serve as an example for a source of metrics.

3.2 Configuration layouts

- **Configuration 1**:

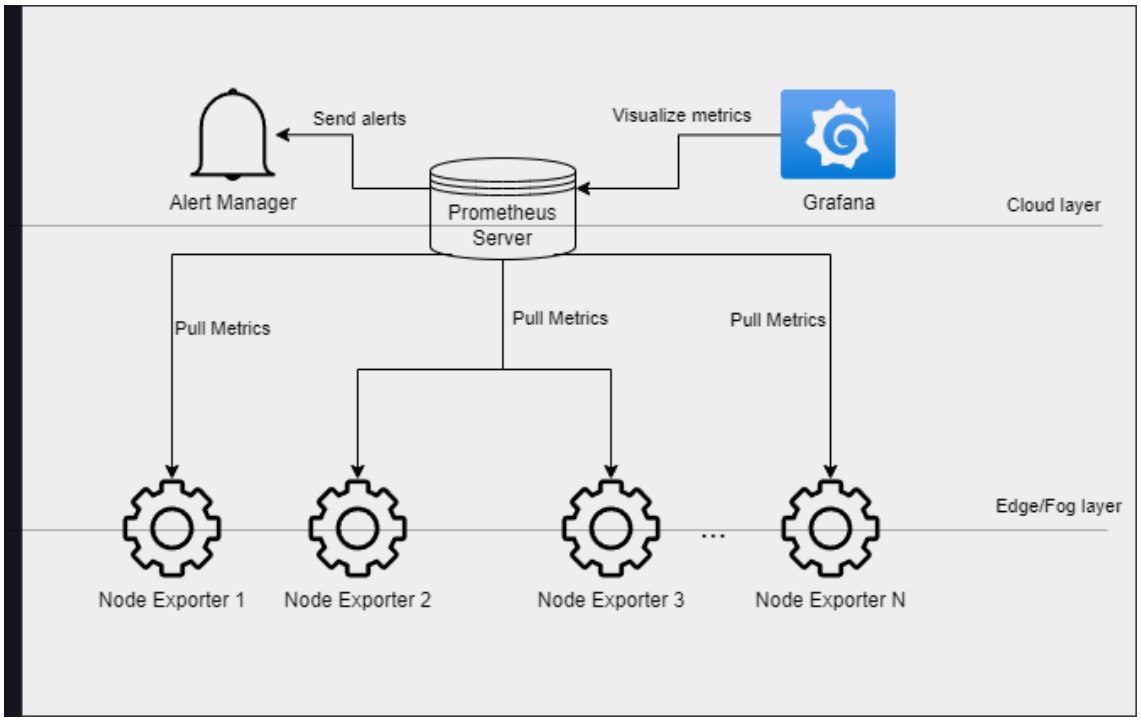


Figure 3.1: Layout of Configuration 1 based on Prometheus.

With this configuration, we designate Prometheus as the cloud storage node. Its job is to gather metrics from every Node Exporter at the edge layer and store them for later examination and display through Grafana dashboards. There is no prior or post-processing done when gathering the metrics. Additionally, the Prometheus Server is responsible for assessing the cloud-level stored metrics, should a rule satisfy the pre-established criteria at runtime, it will transmit this event to the Alert Manager, which will subsequently propagate it to the intended targets.

The key advantages of such a design are that it is easier to set up the monitoring system since there are fewer components to coordinate and, given that the cloud node is scraping the exporters directly, reducing the amount of communication between the monitoring system's nodes resulting in a more efficient use of available resources without the creation of relevant overhead as mentioned in the previous chapter. On the other hand, considering that all metrics are directed towards the cloud node, there will be an increase in resource usage (bandwidth, memory, and CPU). Moreover, alerts will be detected on a later time frame because the Prometheus Server cannot evaluate

the metrics in conjunction with the predefined rules until the metrics are in the cloud storage.

- **Configuration 2**:

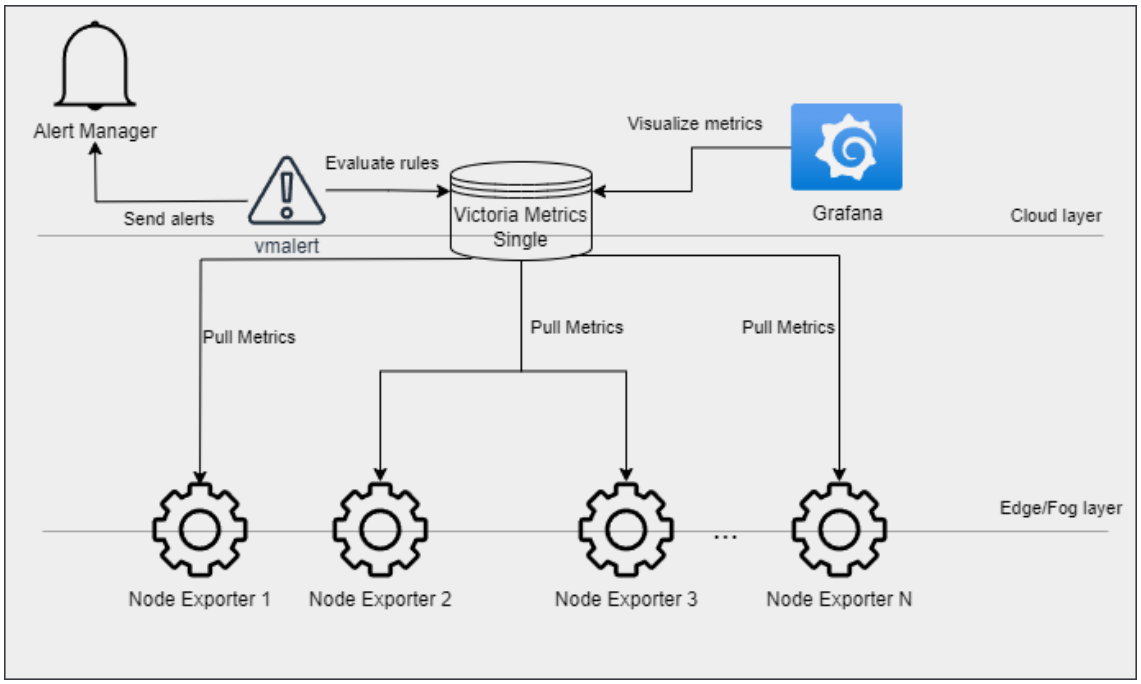


Figure 3.2: Layout of Configuration 2 based on VictoriaMetrics.

In this case VictoriaMetrics Single replaces the Prometheus Server as the central storage node. It is also tasked with scrapping metrics from the edge layer Node Exporters and in turn store them allowing for them to be later evaluated by the system's administrator using the Grafana tool. Contrary to configuration 1 VictoriaMetrics Single does not have the same capabilities as Prometheus Server in terms of rule evaluation, in this case that job is delegated to the vmalert component situated at the cloud layer. It must be connected to a data source, the cloud storage node, and only then it can evaluate its list of previously defined rules and when an alert is detected then it will be sent to the Alert Manager to be propagated.

This configuration shares the same difficulties and setbacks as the previous one, even tho the central node has been replaced with VictoriaMetrics Single as the cloud node, the influx of scrapped metrics will require major bandwidth usage along with increased memory and CPU consumption, despite these constraints, VictoriaMetrics advertises lower memory consumption when comparing to the Prometheus Server that we want to evaluate. The issues regarding late alert detection also carry over to this configuration, even though the alert rule evaluation has been delegated to the vmalert component, it still requires that the metrics reach the cloud node.

- **Configuration 3**:

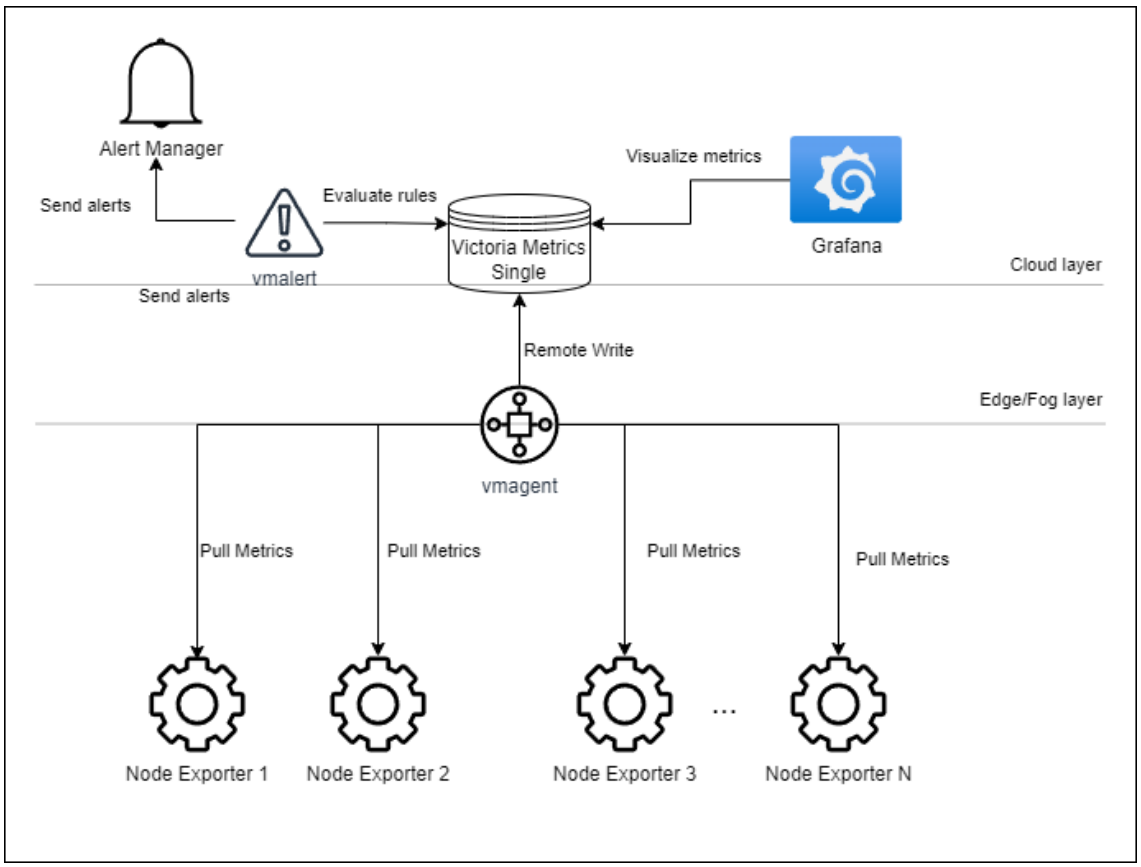


Figure 3.3: Layout of Configuration 3 based on VictoriaMetrics with the addition of the *vmagent* component.

Similarly to the previous configuration, the VictoriaMetrics Single cloud storage node plays the same role, storing scraped metrics for later evaluation and visualization. The Node Exporters remain at the edge layer, but in contrast to earlier configurations, *vmagents* at the edge/fog layer level look to facilitate the task of scraping metrics in the present scenario. Through the remote write protocol, these agents scrape metrics from assigned targets,

and consequently proceed to flush them to the configured data source, in this case, the VictoriaMetrics Single, which is located at the cloud node. Regarding the detection and handling of alerts, this configuration shares the same shortcomings as 1 and 2.

By bringing metric scraping, aggregation, and relabeling closer to the edge layer, this configuration seeks to mitigate the strain associated with the central cloud node through the deployment of additional component nodes like *vmagents*, previously shown to be effective. But in order to take use of the previously mentioned advantages, putting such agents into practice in a system like this requires additional configurations. Even though these new components scrape, temporarily cache, and then push the metrics to the central cloud node, they do not have the capability to be queried on the cached data. On the other hand, vmalert still needs to be evaluating rules on the central cloud node, the data source, so alerts continue to be detected later than what is desirable in a system like this.

- **Configuration 4** :

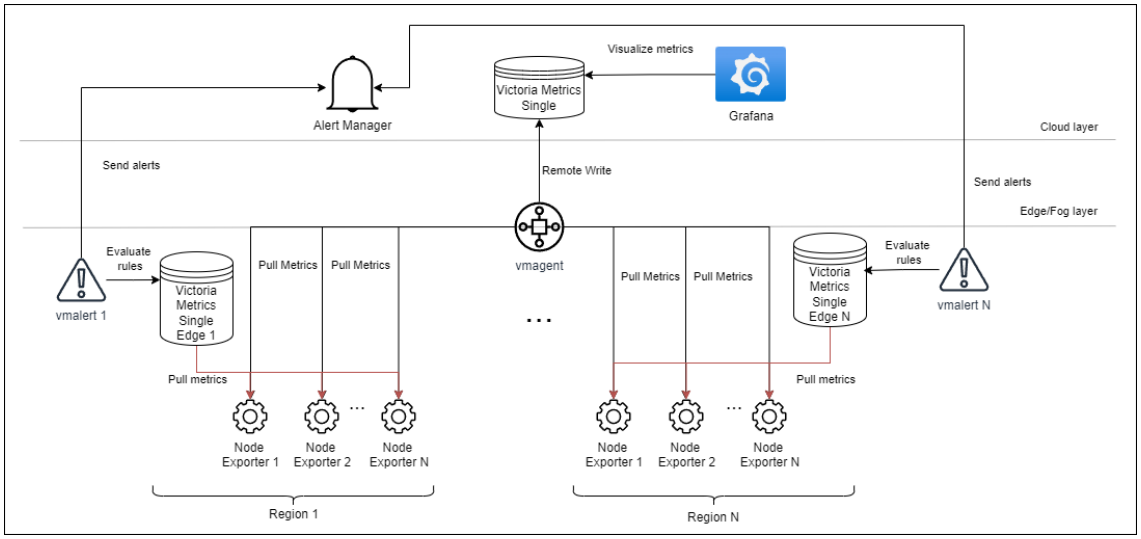


Figure 3.4: Layout of Configuration 4 based on VictoriaMetrics with the addition of edge layer storage.

This configuration is inspired by configuration 3 with the main goal of further reducing the load on the cloud storage node. It uses additional VictoriaMetrics Single instances that will be located at the edge or fog layer, one per monitoring region. These new instances along with the *vmagents* scrape metrics from the Node exporters situated at the edge layer. The *vmagents* in turn will push those metrics into the cloud storage node for further analysis and improved visualization purposes, while the additional VictoriaMetrics Single instances will store them for a shorter time span for alerting purposes. Here the *vmalert* agents are moved from the cloud layer to the edge or fog layer and use these new instances as data sources for alert assessment and propagation.

Looking to further improve on the efforts of the previously mentioned configurations, this configuration relieves the central node from the burden of having to collect and and store all scraped metrics by the *vmagents* or Node Exporters. With the deployment of additional VictoriaMetrics single instances closer to the edge layer and in different regions, the initially established goals are closer to being achieved. In this case the cloud node takes the role of storing the aggregated metrics pushed by the remote write protocol of the *vmagents* and, in turn through the use of Grafana, it allows the system administrator to have a general sense of the system's state. The Node Exporters metrics as a whole are scraped by the added VictoriaMetrics Single nodes at the edge layer and stored for a shorter retention time span.

Given this new layout, we aim at improving the central cloud node's resource usage not only in terms of memory and CPU but mainly in bandwidth due to the reduced influx of network traffic. Regarding alert detection, since we have data sources closer to the edge layer, the *vmalert* component in this configuration can assess and compare its rules with the edge layer stored data in a more timely manner and in turn allowing users to be notified sooner.

Additionally, this layout of components is also intended to tackle the scalability issues associated with large scale monitoring systems since the edge layer VictoriaMetrics Single storage and correspondent *vmagents* are be deployed in different regions, allowing for more efficient resource usage. Finally, in terms of relabeling there are advantages worth mentioning, this layout would allow for metrics coming from a certain region to be grouped and filtered thus reducing the amount of metrics reaching the cloud storage node and giving the system manager an overview of the gathered metrics by monitored region.

4

Evaluation

In this chapter we will compare Prometheus and VictoriaMetrics for the performance evaluation of the previously given scenarios, explaining the advantages, disadvantages, and trade-offs of each approach at a functional level in terms of metric visualization and analysis as well as at a performance level. The purpose of this review is to illustrate architectural considerations and optimization methodologies for building reliable, high- performance monitoring systems that are suited to the particular requirements of contemporary monitoring infrastructures at the cloud and edge levels. Our goal goes beyond simply assessing performance indicators; instead, we also analyze the functional features of the monitoring systems, focusing in particular on the advantages of metric display by using relabeling techniques.

4.1 Methodology

Below, we turn our attention to explaining the detailed processes and setups that were developed in order to carry out thorough testing on the suggested architectures. The 2 dimensions we focused on to carry out the evaluation were:

- **Functional**: with the main focus of clearly displaying the visual advantages of using relabeling and filtering rules by simulation a system in which metrics are being scraped from several different regions. These rules consist of appending to the desired group of metrics a string, a label, that later can be used as a parameter for filtering in relevant queries. This way we are able to illustrate the benefits these measures carry in terms of dealing with large scale monitoring systems when analyzing the scraped metrics using Grafana.

- **Performance**: To better assess the performance of the different layouts in rigorous and consistent manner we developed several docker compose files each implementing one of the previously mentioned configurations, their components, configurations and commands. To retrieve valid metrics regarding these tests a Python script was created to be ran along with the deployment of said docker compose files and retrieve valid and relevant metrics through Dockers Engine API.

4.2 Testing Environment

In order to replicate real-world deployment scenarios in a controlled and reproducible manner, the architecture proposals were tested using a containerized environment. This section identifies and details both the hardware and software components used to perform tests on previously mentioned architecture proposals.

4.2.1 Hardware Specification

All containers were deployed in a self-owned device whose specifications can be found in the following Table 4.1.

Component	Specification
CPU	Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz 4 Cores 8 Threads
RAM	16GB LPDDR3-2133
Main Disk	M.2 PCIe NVMe SSD 512GB
Network Interface	1.73 Gbps

Table 4.1: Device specification

4.2.2 Software Environment

We used Docker to create a testing environment based on containerization to guarantee the accuracy and repeatability of our performance assessments portrayed in Table 4.2.

Component	Specification
OS	Fedora Linux
Kernel Version	v6.6.8
Containerization Engine	Docker v24.0.7
Docker Engine API	v1.45
Docker Compose	v2.21.0

Table 4.2: Software specification

In order to conduct these tests, we assembled docker compose files for each configuration allowing us to deploy numerous containers in a more practical, adaptable, and standardized way for handling complex monitoring architectures. The version of the images used by each component can be observed in Table 4.3.

Component	Image release version
Prometheus	v2.51.0
VictoriaMetrics	v1.93.4
vmagent	v1.93.4
vmalert	v1.93.4
Grafana	v9.2.7
AlertManager	v0.25.0
Node Exporter	v1.7.0

Table 4.3: Container images

As part of our deployment approach, every single component was encapsulated into an own Docker container, creating a modular and isolated environment that was ideal for thorough testing. This method ensured clear boundaries and reduced the likelihood of conflicts by putting every system component within distinct containers. Furthermore, the bridge driver was used to connect these containers over the same network, enabling seamless communication and data exchange.

4.2.3 System Configurations

In order to properly coordinate the configurations described, a structured approach is required, which means that unique YAML property files must be prepared according to the needs of the correspondent component of the monitoring system design. Notably, customized configurations are required for components like VictoriaMetrics Single, Prometheus Server and *vmagent* in order to guarantee optimal performance and seamless integration.

Given their simplicity, the first and second configurations should have comparable characteristics. Important commands such as the *config.file* and the *prom-scrape.config*, respectively, receive emphasis within the docker compose file. These commands set important parameters, which include the scrape interval, specifying how often Prometheus Server and VictoriaMetrics Single scrape metrics from the targets that have been provided in the YAML file's scrape configuration section.

The addition of the *vmagent* component to configurations 3 and 4 represents an important shift in the architecture of the monitoring system by reassigning responsibility for relabeling and metric scraping. Configurations 3 and 4 delegate the scraping and relabeling tasks to the vmagent component, in contrast to configurations 1 and 2, where VictoriaMetrics Single was responsible for both gathering and storing metrics from targets. This fundamental shift alters the cloud node's role by giving it the responsibility of effectively storing metrics for later analysis and visualization. Time-series data from remote sources can be efficiently ingested into VictoriaMetrics instances thanks to the Remote Write Protocol used by *vmagent* component. It can bundle several data points into a single request due to its batched ingestion features, which reduces the amount of disk IO, network bandwidth, and disk space consumed. New commands in the Docker compose file are required in order to enable *vmagent* to be integrated into the monitoring ecosystem. Specifically, the *remoteWrite.url* command must be supplied since it indicates the storage endpoint to which the metrics that are scraped will be delivered using the remote write protocol.

Moreover, the command *remoteWrite.flushInterval* holds great significance in determining how frequently the gathered metrics are sent to the storage node. Administrators may optimize the performance of the monitoring system and guarantee that no crucial metric data is lost in transit by precisely selecting the flush interval. This allows them to strike a compromise between resource efficiency and timely data transmission.

Extra fields are added to the *scraping configs* property of the YAML file for

vmagent setups in order to enable more complex features like streaming aggregation and relabeling. These improvements play a crucial role in maximizing the effectiveness, economy of resources, and scalability of the monitoring system.

First off, adding the labels field to the *scrape configs* property marks a significant advancement in improving the structure and readability of incoming metric data. This field lets administrators give unique labels to metrics that come from particular targets. Administrators enable more insightful analysis and visualization by linking metrics to predetermined labels. The ability to relabel data not only improves its interpretability but also sets the stage for further, more complex filtering, grouping, and analytic processes.

Furthermore, the *scrape configs* property of the YAML file now includes a stream aggregation field, demonstrating a proactive attempt to optimizing resource utilization and minimizing data transfer overhead. In order to reduce the amount of data sent to the cloud node, the monitoring system aims to aggregate metric data at the closest point to the edge layer as possible. When a setup makes use of the *avg* output option, multiple data points that with the same name are combined into a single aggregated value by having the monitoring system determine the average value of a certain metric over the number of targets. Configurations like this reduce network bandwidth utilization and resource consumption significantly by aggregating metric data at the source and delivering just the necessary aggregated values. This optimization technique improves the overall scalability and performance of the monitoring infrastructure while also reducing the load on the cloud node, especially in distributed and resource-constrained scenarios.

Configuration 4 requires extra configurations specifically designed to handle the addition of new storage nodes that are placed closer to the edge layer. There is a need to build a unique YAML file with details on factors like scrape intervals and scraping targets, similar to what happened in configurations 1 and 2.

On the other hand, configuration 4's inclusion of edge nodes brings special considerations meant to maximize data transmission efficiency and minimize alarm detection times. The storage configurations of the edge nodes are strategically changed to achieve these goals. Interestingly, these edge nodes are configured with the shortest storage retention period of 24 hours. This purposeful decision represents a trade-off between data freshness and storage capacity, giving priority to signal detection quickness and real-time responsiveness over long-term data preservation. The latter would be reserved for the cloud storage node.

4.3 Functional Evaluation

Whenever relabeling rules were not implemented to *vmagents*, as configurations 1 and 2 showed, it was challenging and puzzling to analyze the scraped metrics. The lack of organized labeling made it difficult to distinguish where the measurements came from and how they were used, which further negatively impacted the analytic process. System monitoring managers would face a maze of unstructured data that lacked context, making it challenging for them to recognize trends, pinpoint irregularities, or link metrics to specific services or components as percieved from observing both Figure 4.1 and Figure 4.2.

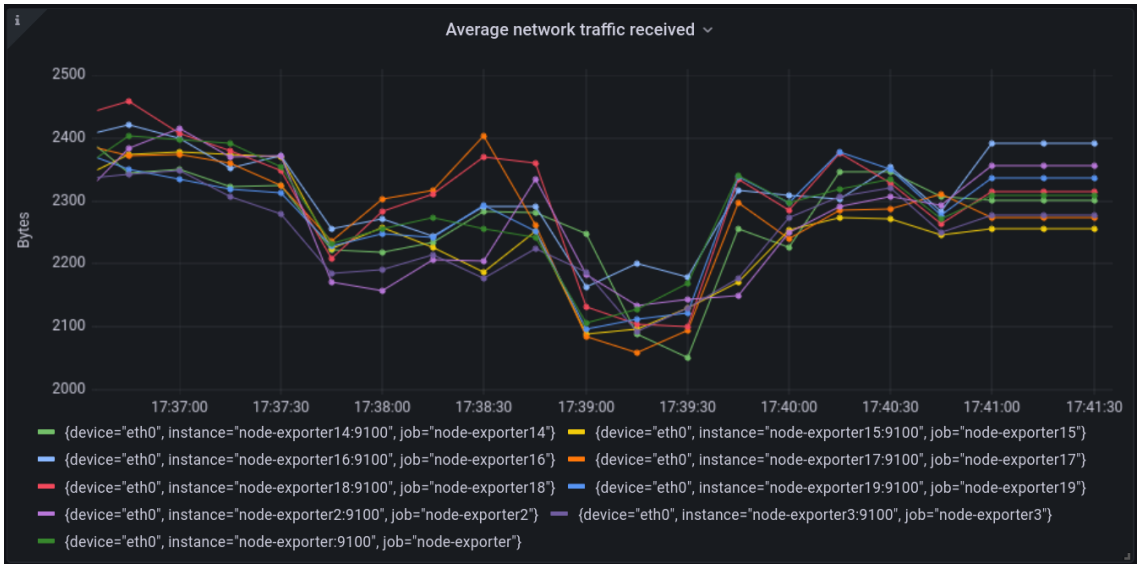


Figure 4.1: Dashboard without relabeling and filtering

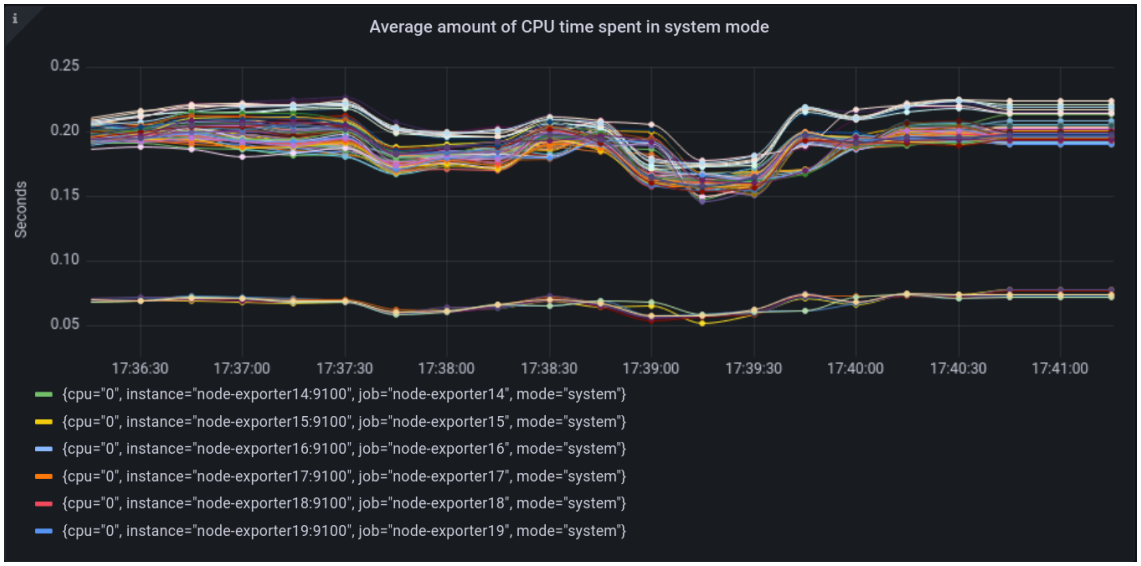


Figure 4.2: Dashboard without relabeling and filtering

Through configuration 4 in which we simulated one VictoriaMetrics Single instance as the sole cloud node and 3 regions(Europe, America and Asia) each with a set of 3 Node Exporters, 1 *vmagent* and another VictoriaMetrics Single instance at the edge layer we were able to implement relabeling rules that would group metrics either in regards with the region they came from as well as the service they were associated with. The implementation of relabeling procedures brought clarity and efficiency to the monitoring infrastructure, greatly improving the viewing experience due to the fact that the dashboards used were now much simplified. Systems managers are able to navigate the monitoring environment with increased accuracy due to this.

Metric relabeling made it possible for users to create queries that focused on certain labels, allowing for more complex data analysis and interpretation. With this newfound control showcased in Figure 4.3 and Figure 4.4, systems managers can customize their queries to retrieve relevant information, such as tracking resource usage in various regions such, keeping an eye on the functionality of vital services, or spotting anomalies that could lead to issues.

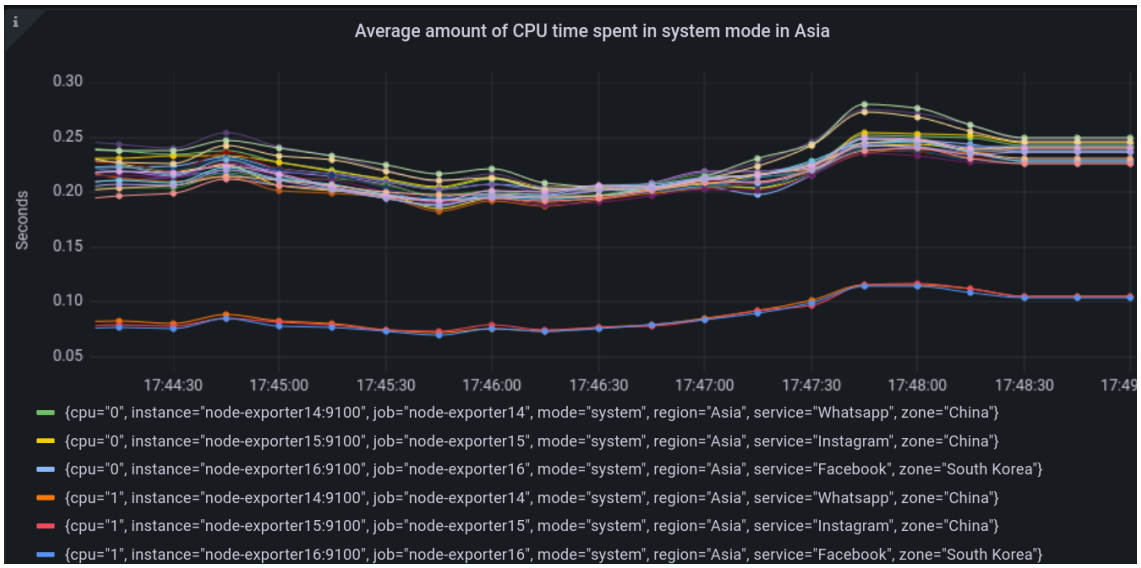


Figure 4.3: Dashboard with relabeling and filtering

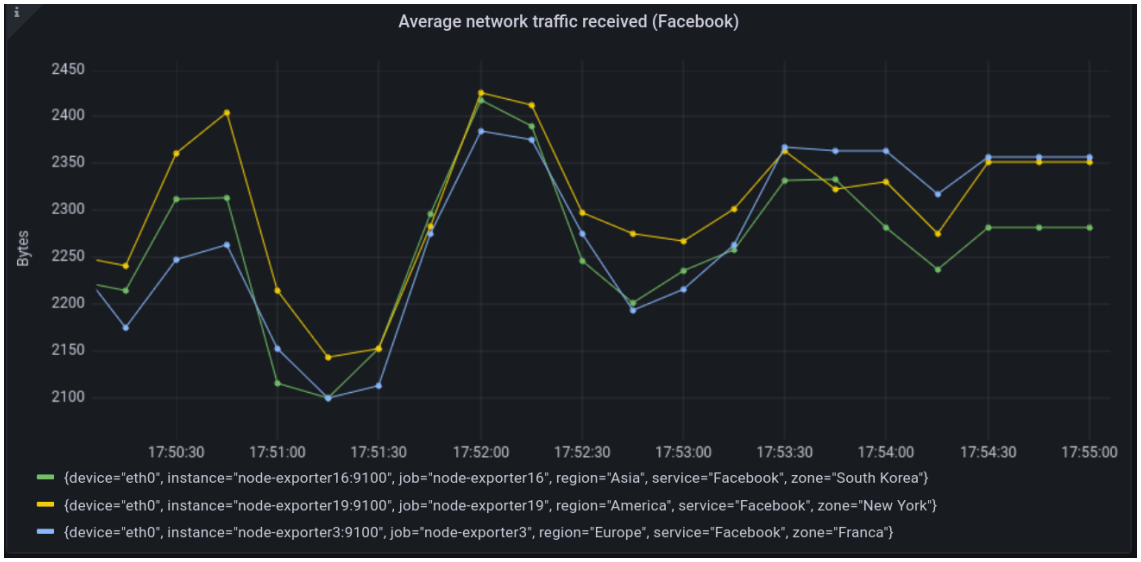


Figure 4.4: Dashboard with relabeling and filtering

4.4 Performance Evaluation

In order to assess the strong points and short comings of the different configurations, comparison scenarios were drawn using the time frames of 1, 15, 30 and 60 seconds for the scraping interval property. These will exemplify the resource usage for strict timely monitoring (for best real time alerts and visualization) vs delayed timed when trying to spare and save resources, particularly when in large scale deployments. Strict memory limitations were placed on every container in order to replicate relevant results and precisely evaluate the performance of the architectural approaches. To be more precise, each container was limited to 100 Megabytes of system memory, simulating the resource limitations that are frequently seen in realistic settings. This targeted restriction was put in place to evaluate how well the monitoring system's components functioned in memory-constrained settings. In addition to showcasing the system's raw performance capabilities, the test results we obtained improved the validity and application of our evaluation by exposing the system's resilience and flexibility in resource-constrained circumstances. With regard to CPU and network use, all containers were granted complete access to the current system potential as mentioned on Table 4.1 .

To evaluate the performance of the different proposals and its components we had to established what metrics we needed to take into consideration. These consist of % of CPU and Memory usage and amount of kilobytes both sent and received.by the network.

Two methods were taken into consideration to extract such metrics from

the various architectural proposals: either use Docker's API or build a parallel monitoring system within the same device to obtain the metrics from the various containers into a dedicated storage instance. As the second option would impact the benchmarking procedure and strain the available resources, a script was created utilizing the Docker Engine API to interact with the Docker daemon directly.

A Python script then developed for metric extraction by orchestrating periodic requests to the `/{api-version}/containers/{container-id}/stats?stream=false` endpoint, utilizing Docker's API to retrieve essential performance data. Furthermore, in order to maximize the effectiveness of data retrieval, the query parameter was purposefully set to false, ensuring that the statistics were gathered only once for each request. As a result of this approach, the script was able to obtain real-time metrics from every single container in a systematic way within the stipulated 10-minute time limit, giving a complete overview of the the system's performance. Utilising the multiprocessing package in Python, the script applied parallel processing techniques to maximize efficiency upon data retrieval. To be more precise, a Pool object from said package was created with as many workers as there were containers deployed. Every worker in the multiprocessing Pool was tasked with submitting queries to the Docker API endpoint for a specific container. Each worker collected metrics at the conclusion of the time frame specified, which were then entered into a dictionary object.

During some trial runs we noticed that after the, `docker compose up -d`, command was used to launch the docker compose file there was a period of time in which the containers were 'warming up' during this period, the containers underwent an initialization processes, and the metrics generated did not accurately reflect the steady-state operational conditions of the system, to combat this, prior to starting the API request process, a 20 second grace time was added to the script. By purposefully delaying the deployment, enough time was given to the containers to stabilize and become operationally ready, guaranteeing that the metrics recorded after the grace period accurately reflected the true performance characteristics of the system. The final stage of the script consists on iterating through the dictionary values and calculate the maximum, minimum and average % of CPU and memory usage along with total amount of kilobytes sent and received. The algebraic formulas used for calculating the established metrics are displayed in Table 4.4.[1]

[1]Both incoming and outgoing network traffic metrics were divided by 100 in order to convert from Bytes to Kilobytes

Table 4.4: Algebraic formula

Metric	Expression
% CPU Usage	$\frac{CPU\_delta}{system\_CPU\_delta} * online\_CPUs * 100$
% Memory Usage	$\frac{memory\_stats\_used}{memory\_stats\_limit} * 100$
Kilobytes received	$\frac{rx\_bytes}{100}$
Kilobytes sent	$\frac{tx\_bytes}{100}$

- **Scenario 1**: *The benefits of the vmagent component*

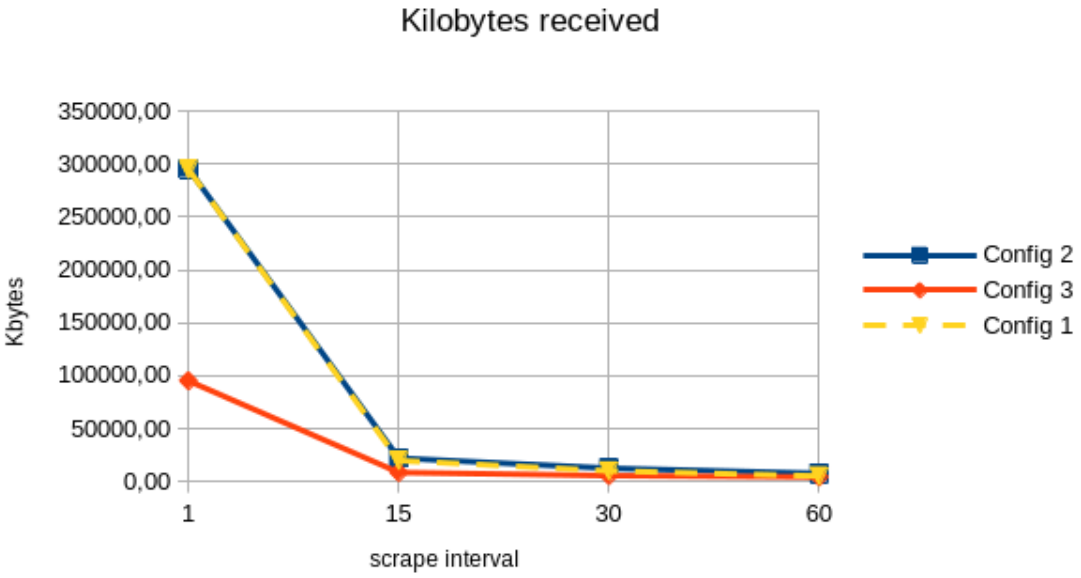


Figure 4.5: Received kilobytes for configuration 1, 2 and 3.

In the present scenario, our first objectives are to investigate the effect of the *vmagent* component on resource consumption at a cloud storage node level by comparing configuration 2 and 3 and also the performance difference between the two presented metric storage solutions, Prometheus Server and VictoriaMetrics Single, using more simplistic architectural configurations, 1 and 2, respectively, with 3 Node Exporters as source of metric extraction.

From observing Figure 4.5 above we are able to assess that in terms of amount of data received at the cloud level the solution that implements *vmagent* to scrape and aggregate metrics, configuration 3, is relieved of a load of more than 3x the Kilobytes when compared to both configuration 1 and 2 at the scrape interval of 1 second. This was possible due to the aggregation capabilities of the *vmagent* component. The remaining time

frames show that in spite of showing similar values graphicaly there is still a clear advantage in using a *vmagent*.

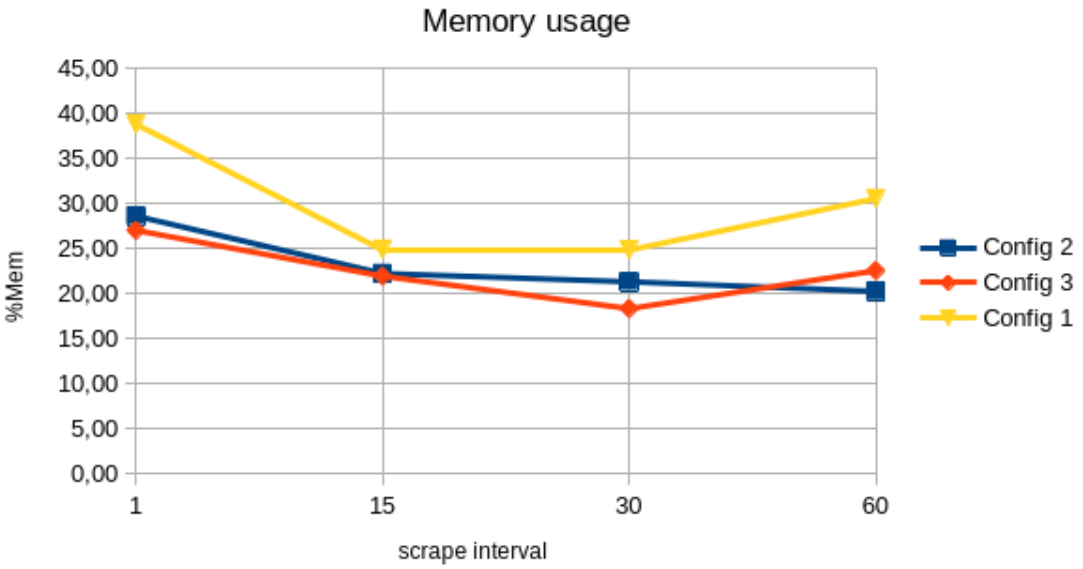


Figure 4.6: % Memory usage for configuration 1, 2 and 3.

In terms of % memory usage the results shown on Figure 4.5 allows the conclusion that both VictoriaMetrics configurations required less of the systems resources when comparing to configuration 1 where Prometheus is used. Nonetheless, the trend remains on the majority of time scrape intervals with the use of the *vmagent* component handling loads that otherwise would be delegated to the central cloud node.

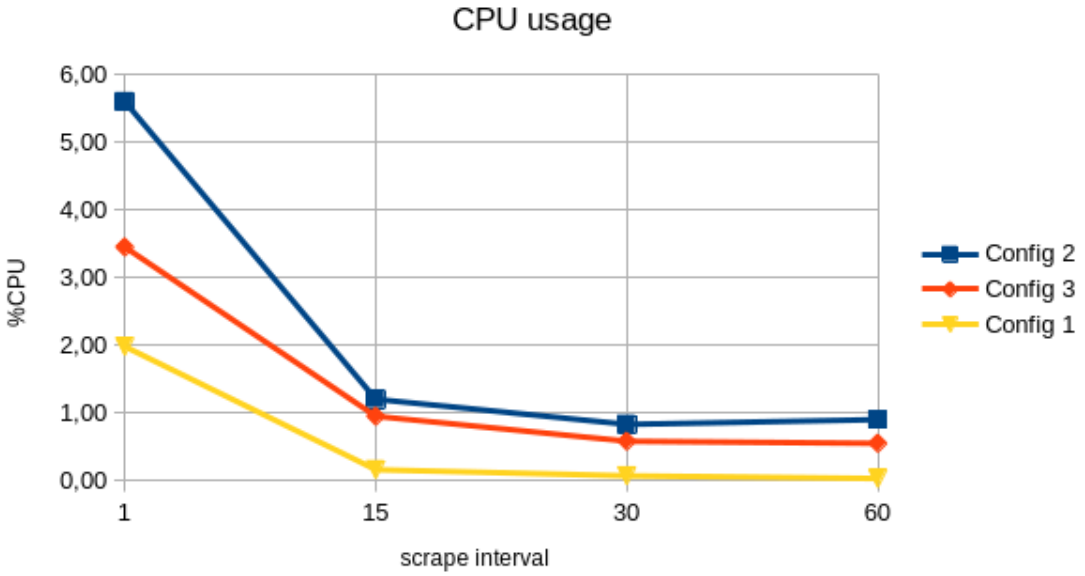


Figure 4.7: % CPU consumption for Configurations 1, 2 and 3.

When analyzing the % of CPU usage on Figure 4.7 we are able to conclude that, across all scrape intervals, both configurations using VictoriaMetrics Single instances as the cloud storage node proved to require more of the systems CPU than configuration 1 contradicting the previously set trend.

From this scenario we are able to withdraw two conclusions, first, making use of the micro service architecture of VictoriaMetrics and consequently its agents provides an overall increase in monitoring performance given that the overall resource usage proved to be inferior to its *vmagentless* counterpart, and secondly despite using slightly more % of the systems CPU, Figure 4.7, the overall benefits of using VictoriaMetrics outweigh Prometheus, thus being the best option in terms of reducing the amount of resources used by the cloud node.

- **Scenario 2**: *The cloud benefits of implementing storage nodes at the edge layer*

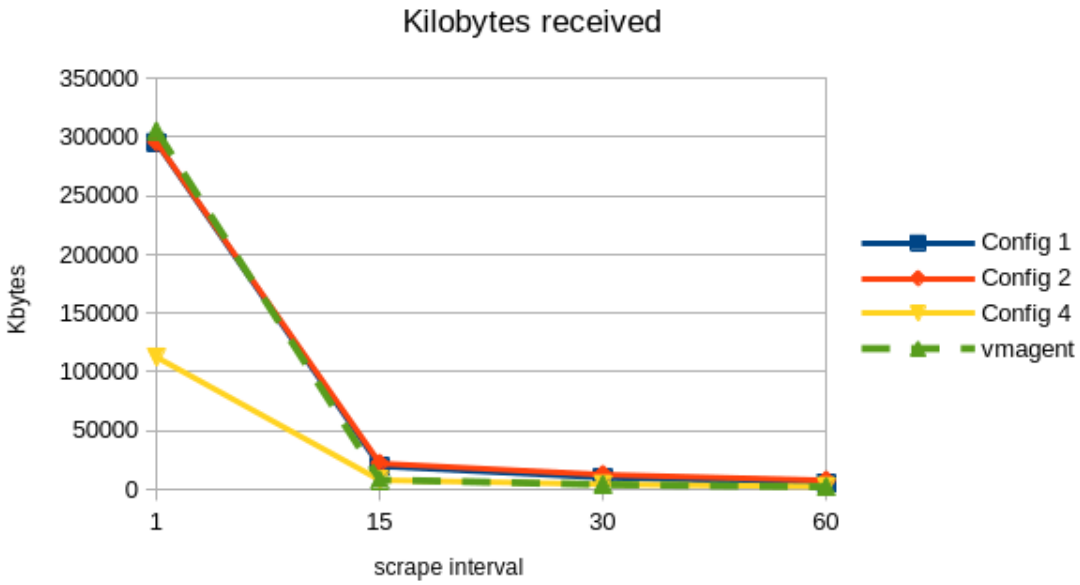


Figure 4.8: Received Kilobytes for configuration 1, 2 and 4.

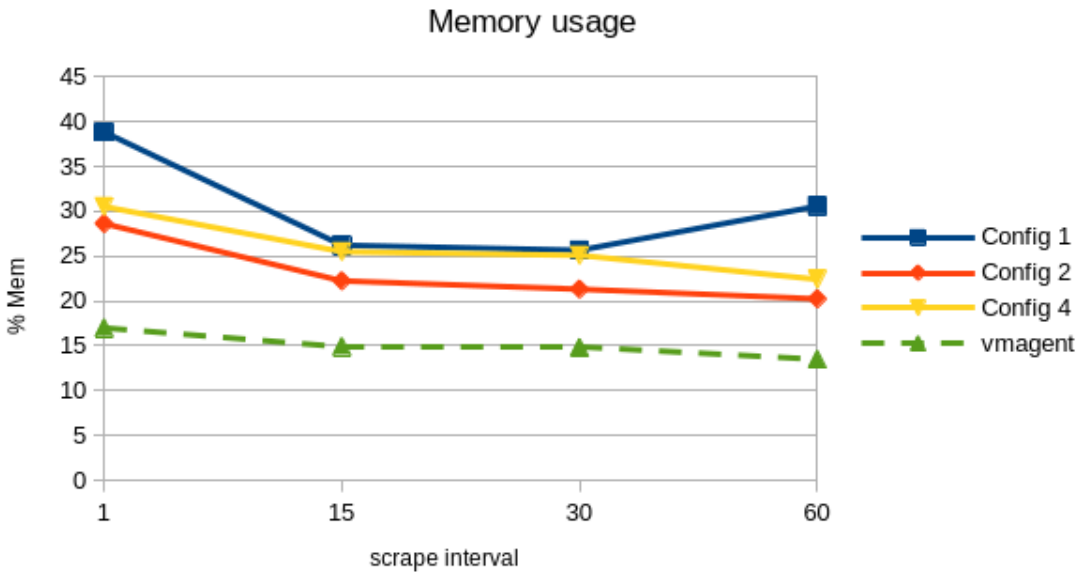


Figure 4.9: % Memory consumption for configuration 1, 2 and 4.

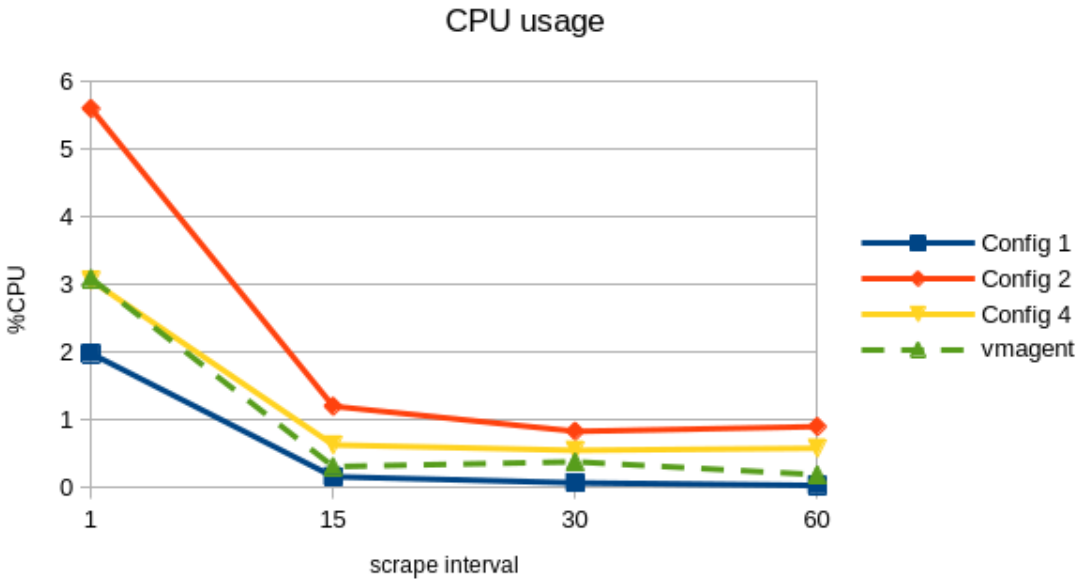


Figure 4.10: % CPU consumption for Configurations 1, 2 and 4.

Here we aim at providing proof that makes the implementation of additional VictoriaMetrics Single instances with shorter metric retention time the right decision in the path of achieving optimal resource usage by the cloud node. Present in the above graphs are test results from configurations 1, 2 and 4. The first and second configuration do not include the *vmagent* component in their layout as well as the additional storage node only present in configuration 4.

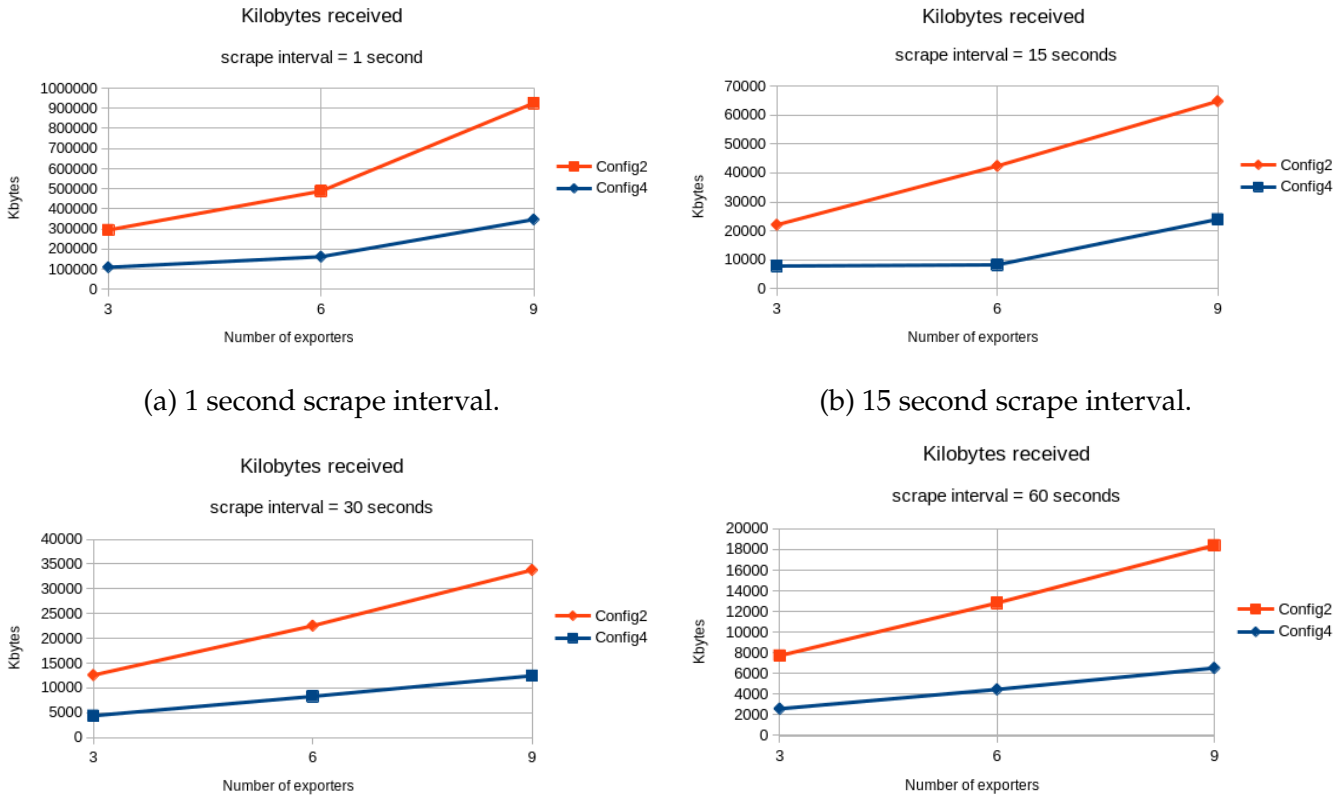
Comparatively to the previous scenario, both configurations 1 and 2 share similar values in regards to the amount of Kilobytes received, since there can not be any sort of metric filtering or aggregating for rigorous alert detection purposes, the cloud nodes present in either configuration must store all values received. In contrast, configuration 4 now takes advantage of its additional component to tackle this issue not only by absorbing the load that would go to the cloud node but also by improving on the time it takes to detect and trigger an alert. Metric aggregation and relabeling is mandatory to reduce, even more, the amount of resources used by the cloud node, the system administrator in turn would only need to have an overview of the systems performance thus not being required that Grafana gets fed the full range of metrics using a shorter scrape interval.

By looking at Figure 4.8 we are able to observe that the yellow line, which represents the resource usage of a cloud node with subsequent edge layer storage node, presents a lower amount of Kilobytes received when comparing

to its configuration counterparts. The outstanding load that would have went to said cloud node can be observed through the dotted green line, it represents the amount of metrics received and consequently aggregated by the *vmagent* thus not reaching the cloud node. On the other hand % of memory usage results show that VictoriaMetrics Single cloud nodes share similar values when the scrape interval is 1 and 60 seconds but always in favour of configuration 2.

Given these results and regarding the main objective of reducing bandwidth usage along with % CPU and memory while maintaining fast alert detection and triggering we concluded the benefits of a monitoring solution based on configuration 4 overcome the downsides that come with it, namely the overhead in terms of system configurations.

- **Scenario 3**: *Impact of node scalability in cloud resource usage*



(a) 1 second scrape interval.

(b) 15 second scrape interval.

(c) 30 second scrape interval.

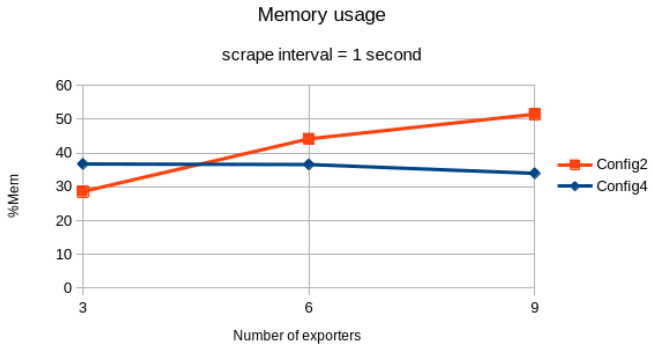
(d) 60 second scrape interval.

Figure 4.11: Impact of additional targets on Kilobytes received.

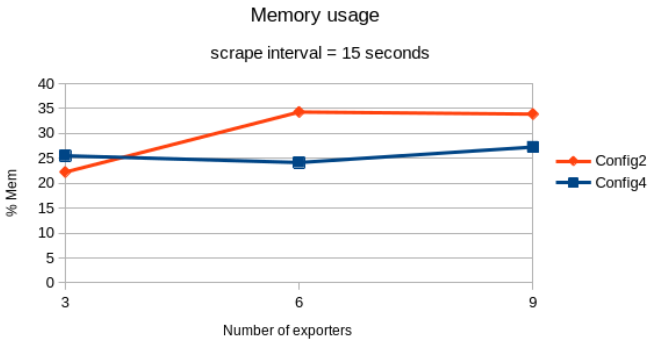
In this final scenario, we examine the challenges associated with large-scale monitoring systems and investigate how expanding the number of scrape targets affects cloud nodes.

Increasing the number of targets causes cloud nodes to face higher data ingestion rates, greater processing overhead, and higher memory usage. These factors collectively put a strain on the underlying infrastructure and can cause bottlenecks in terms of scalability, resource contention, and performance degradation.

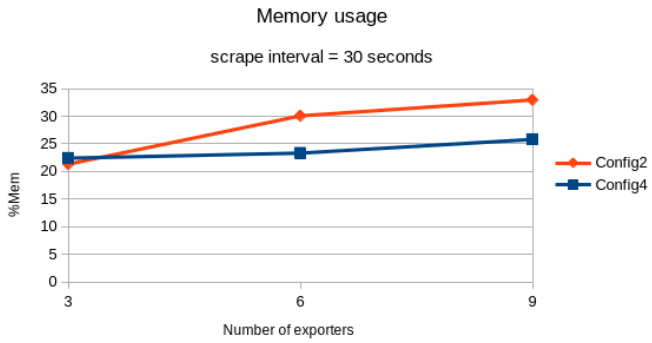
Configurations 2 and 4 where taken into consideration in order to better represent this issue. By observing the kilobytes received graphics above in Figure 4.11 we are able to conclude that, across all scrape intervals, when the number of Node Exporters the amount of kilobytes received by the cloud node at configuration 4 is always lower than configuration 2. This can be explained by the fact that the *vmagents* took the role of aggregating scraped metrics in the region they were deployed in.



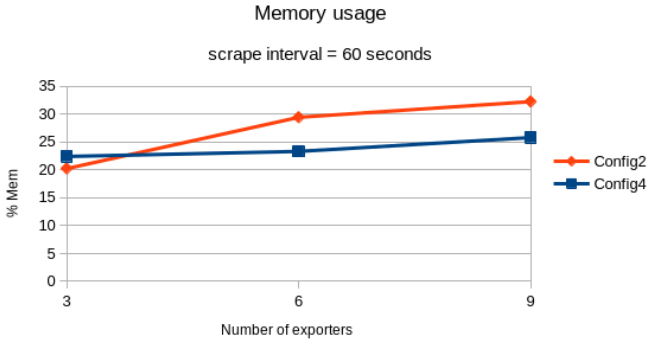
(a) 1 second scrape interval.



(b) 15 second scrape interval.



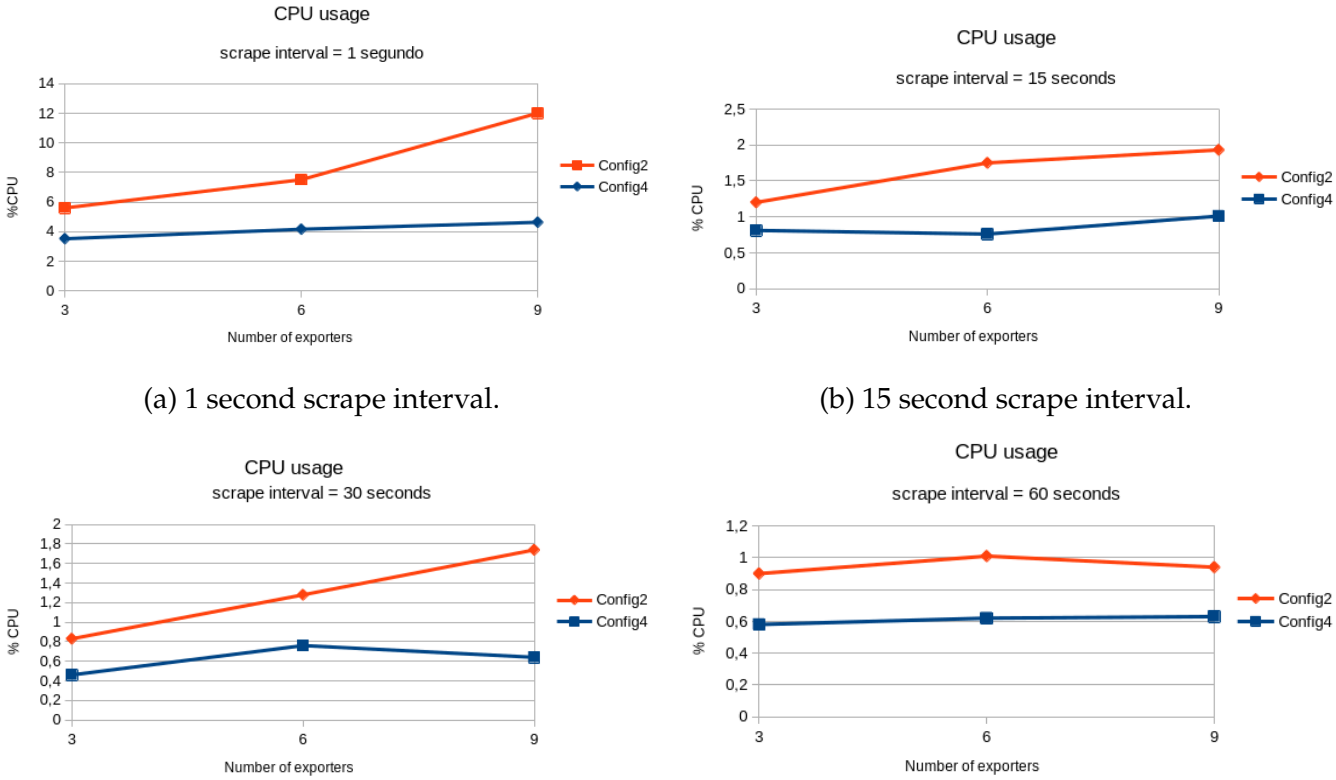
(c) 30 second scrape interval.



(d) 60 second scrape interval.

Figure 4.12: Impact of additional targets on % Memory consumption received.

In terms of % of memory usage a similar result takes place in Figure 4.12, when looking into the different scrape intervals we can observe that when the number of Node Exporters is three configuration 2 has the edge of its higher overhead configuration counterpart, but when the number of nodes increases to six and then nine the results shift and configuration 4 with its several regions will in turn consume fewer memory.



(a) 1 second scrape interval.

(b) 15 second scrape interval.

(c) 30 second scrape interval.

(d) 60 second scrape interval.

Figure 4.13: Impact of additional targets on % CPU consumption received.

Similarly to the first set of graphs, when overviewing the % of CPU usage across the different scrape intervals, Figure 4.13, we are able to draw the same conclusions, when the number of targets increases so does the % of CPU usage, but due to its layout and component orchestration configuration 4 is able to better manage incoming loads presenting improved results when comparing it to configuration 2.

We are able to then conclude that a monitoring architecture that decentralizes metric scraping and implements a micro service like structure, for instance, deploying several *vmagents* per existing region, significantly improves handling the growth in the number of scraping targets. Furthermore it allows for increasing the scraping interval of the agents that push metrics into the cloud across different regions, reducing resource utilization by the cloud node whilst not compromising rule evaluation since this would be taken care of by the edge layer storage, with shorter scrape intervals, allowing for a shorter alert detection and subsequent propagation with lower latency, implemented in configuration 4.

5

Conclusion and Future Work

With this dissertation we set out to solve the issues connected with monitoring large amounts of targets in edge and cloud systems aiming at creating an architecture that could both reduce the amount of resources consumed by the cloud nodes, CPU and memory usage along with bandwidth, while also maintaining a low latency regarding alert detection and propagation.

During the research phase two systems stood out as the leading candidates for assembling an ideal architectural solution, Prometheus and VictoriaMetrics. After the creation and configuration of different layouts several containerized tests were ran in order to assess their performance in a smaller scale environment so as to be able to analyze their results and extrapolate them for real-life monitoring scenarios.

After the evaluation phase, we were able to conclude that in spite of requiring a larger configuration overhead and orchestration of different components, the optimal solution was configuration 4, with VictoriaMetrics Single as its cloud node storage component. Given that such design allows to release a considerable amount of data from the cloud layer to its edge components our tests showed that by choosing such architecture were able to reduce the incoming metrics by 38%, considering the most demanding scrape interval of 1 second, when compared with other proposed solutions, whilst not compromising CPU and memory usage through the use of the *vmagent* component that relieved the cloud node from metric scraping duties.

We were also able to showcase the benefits and downsides of increasing the scrape interval, for instance from 1 second to 15 seconds through scenario 2. In one hand we improved on the % of resources used. For configuration 1, 2 and 4, on average, the amount of kilobytes received decreased 92% severely reducing the bandwidth usage by the cloud node but on the other hand this means that the first and second configuration will be lacking tremendous amount of metrics

needed to perform alert detection rule evaluation. In case of configuration 4 this problem is not as impactful given that the full amount of metrics will be scraped into the edge layer storage.

Moreover, by analyzing the results gathered on scenario 3 we were able to observe that increasing the number of target nodes has a great impact on the resource usage of both centralized and distributed architectures, configuration 2 and 4 respectively. For instance, when we increase the number of nodes from 6 to 9, in terms of kilobytes received, on average through the different scrape intervals, we observed that there was an increase of 40% for configuration 2 and 24% for configuration 4. We were than able to conclude that even though there is an increase in amount of network traffic in both situations, configuration 4 stands out as being able to better handle such scenario as it displayed better results than configuration 2 further proving that a decentralized approach is more effective in terms of monitoring large amount of targets.

After testing on this matter we were able to record positive results regarding the predefined evaluation metrics. This solution targeted the main objectives of this dissertation by allowing early alert detection and propagation to be possible while also tackling the scalability issues that start to unravel once the number of targets increases. These accomplishments were achieved by deploying edge level small retention VictoriaMetrics Single storage nodes in different regions thus spreading the load to the edge of the network and across multiple nodes releasing the computational burden from the central cloud node. Additionally, as mentioned in functional evaluation we were able to associate custom metrics with incoming scraped metrics tackling the final objective of implementing a solution that enables system administrators to pinpoint in a more efficient way potential issues and discover trends that may optimize the monitoring systems layout and component layout.

Finally, even though the objectives for this dissertation were achieved there is still room for improvement and optimization. Regarding the architectures of such monitoring systems, the development of a new feature for the *vmagent* component would greatly increase the orchestration of said systems while also aiming at simplifying deployment and configuration. If the *vmagent* component had a feature that would allow its scraped metrics to be queried using the same mechanism as the VictoriaMetrics Single instances it would allow for faster alert detection and propagation as well as further reduce the monitoring systems complexity in terms of components deployed and subsequent configurations. Furthermore, a system manager overviewing the incoming metrics could observe that a shorter scrape interval would be beneficial in terms of resource usage. To remediate this

situation, instead of having to re-deploy the whole system a valuable new feature would allow system administrators to adjust configuration values during runtime through the use of an API.

Bibliography

[1] Á. Brandón et al. "FMonE: A Flexible Monitoring Solution at the Edge". In: *Wirel. Commun. Mob. Comput.* 2018 (2018-01). ISSN: 1530-8669. DOI: 10.1155/2018/2068278. URL: https://doi.org/10.1155/2018/2068278 (cit. on p. 17).

[2] R. P. Centelles et al. "REDEMON: Resilient Decentralized Monitoring System for Edge Infrastructures". In: *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. 2020, pp. 91–100. DOI: 10.1109/CCGrid49817.2020.00-84 (cit. on p. 18).

[3] Google. *Google-cadvisor: Analyzes resource usage and performance characteristics of running containers.* URL: https://github.com/google/cadvisor (cit. on p. 16).

[4] *Grafana*. URL: https://grafana.com/docs/grafana/ (cit. on p. 11).

[5] M. Großmann and C. Schenk. "A Comparison of Monitoring Approaches for Virtualized Services at the Network Edge". In: *2018 International Conference on Internet of Things, Embedded Systems and Communications (IINTEC)*. 2018, pp. 85–90. DOI: 10.1109/IINTEC.2018.8695277 (cit. on p. 16).

[6] C. B. Hauser and S. Wesner. "Reviewing Cloud Monitoring: Towards Cloud Resource Profiling". In: *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. 2018, pp. 678–685. DOI: 10.1109/CLOUD.2018.00093 (cit. on p. 19).

[7] T. Jackson. *Promxy*. 2017. URL: https://github.com/jacksontj/promxy (cit. on p. 8).

[8] S. Kum et al. "Optimization of Edge Resources for Deep Learning Application with Batch and Model Management". In: *Sensors* 22.17 (2022). ISSN:

1424-8220. DOI: 10.3390/s22176717. URL: https://www.mdpi.com/1424-8220/22/17/6717 (cit. on p. 19).

[9] J. M. Lourenço. *The NOVAthesis LaTeX Template User's Manual*. NOVA University Lisbon. 2021. URL: https://github.com/joaomlourenco/novathesis/raw/main/template.pdf (cit. on p. ii).

[10] *M/Monit*. URL: https://mmonit.com/documentation/ (cit. on p. 14).

[11] Prometheus. *Prometheus - Monitoring System amp; Time Series Database*. URL: https://prometheus.io/ (cit. on p. 6).

[12] H. J. Syed et al. "CloudProcMon: A Non-Intrusive Cloud Monitoring Framework". In: *IEEE Access* 6 (2018), pp. 44591–44606. DOI: 10.1109/ACCESS.2018.2864573 (cit. on p. 21).

[13] *The OpenNMS Group*. URL: https://www.opennms.com/ (cit. on p. 12).

[14] A. Valialkin. *Billy: How victoriametrics deals with more than 500 billion rows*. 2020-04. URL: https://valyala.medium.com/billy-how-victoriametrics-deals-with-more-than-500-billion-rows-e82ff8f725da (cit. on p. 12).

[15] A. Valialkin. *Measuring vertical scalability for time series databases in google cloud*. 2019-06. URL: https://valyala.medium.com/measuring-vertical-scalability-for-time-series-databases-in-google-cloud-92550d78d8ae (cit. on p. 11).

[16] A. Valialkin. *Prometheus vs Victoriametrics Benchmark on node_exporter metrics*. 2022-01. URL: https://valyala.medium.com/prometheus-vs-victoriametrics-benchmark-on-node-exporter-metrics-4ca29c75590f (cit. on p. 12).

[17] A. Valialkin. *When size matters-benchmarking Victoriametrics vs timescale and influxdb*. 2019-05. URL: https://valyala.medium.com/when-size-matters-benchmarking-victoriametrics-vs-timescale-and-influxdb-6035811952d4 (cit. on p. 11).

[18] *VictoriaMetrics*. URL: https://victoriametrics.com/ (cit. on p. 9).

[19] *Xarxa de telecomunicacions de Comuns, Oberta, Lliure I neutral*. URL: https://guifi.net/ (cit. on p. 18).