PROJECT

Multi-tenant cybersecurity operations centre designed
as an open cloud service with machine learning elements

Linked to work package KPB1 – Design of the virtualization solution

Linked to output
V3 – Description of the selected solution meeting the specified criteria

type – documentation

Monitoring of an IaaS CC platform built on OpenStack – experiment

CONTENTS

Introduction **5**
1 Analysis **6**
1.1 Description of the topic 6
1.2 Motivation for cloud monitoring 6
1.2.1 Resource planning 6
1.2.2 Resource management 6
1.2.3 SLA management 7
1.2.4 Billing 7
1.2.5 Troubleshooting 7
1.3 Basic concepts for cloud monitoring 7
1.3.1 Cloud layers 7
1.3.2 Monitoring levels 8
1.3.3 Monitoring tests and collected metrics 8
1.3.4 Monitoring views 9
1.4 Centralized collection of log records 11
1.5 Monitoring system of the *OpenStack* cloud platform built on the Prometheus solution 12
1.5.1 Prometheus 12
1.5.2 Advantages of the solution 13
1.5.3 Prometheus and SNMP 13
1.5.4 Visualization 14
1.5.5 Possible deployment method 14
2 Results of the work – experimental deployment of Prometheus + Grafana 16
2.1 Installation and deployment of the ingress router 16
2.1.1 Installation of Prometheus 16
2.1.2 Prometheus as a service 18
2.1.3 Installation of the Grafana visualization tool 19
2.1.4 Configuration of and access to Prometheus 20
2.1.5 Configuration of and access to Grafana 22
2.1.6 Deployment of the *OpenStack* exporter 24
2.1.7 Configuration of a new source for Prometheus 27
2.1.8 Connecting Grafana+Prometheus and deploying a panel for openstack-exporter 28
3 Deployment of a functional solution 32
3.1 System requirements and components 33
3.2 Monitoring the state of the physical infrastructure 33
3.3 Monitoring the state of VMs in the CC platform 34
3.3.1 Deployment of the solution 35
Conclusion 37
References 38

List of figures

Figure 1.1 Combination of the view and the level of cloud monitoring 11
Figure 1.2 Prometheus architecture 13
Figure 2.1 Prometheus download process 17
Figure 2.2 show folder with Prometheus.tar 17
Figure 2.3 Extracting the Prometheus archive 18
Figure 2.4 Contents of the Prometheus folder 18
Figure 2.5 Verifying the creation of the prometheus.service file 18
Figure 2.6 Prometheus expression browser 21
Figure 2.7 Initial data search 22
Figure 2.8 Returning the number of responses obtained for the query 22
Figure 2.9 Grafana login screen 23
Figure 2.10 Grafana home screen 24
Figure 2.11 Initial test of openstack-exporter 27
Figure 2.12 Prometheus status->targets 27
Figure 2.13 Initial state of openstack-exporter 27
Figure 2.14 Working state of openstack-exporter 28
Figure 2.15 Grafana Data sources 29
Figure 2.16 Importing the monitoring screen p1 30
Figure 2.17 OpenStack exporter dashboard 31
Figure 2.18 Import screen for the dashboard 31
Figure 2.19 Topology of the functional deployment 32
Figure 2.20 Node-Exporterr monitoring screen 34

INTRODUCTION

The world of information technology is constantly evolving. In recent years, this field has been dominated by the concept of cloud computig (CC). The main element that enabled the emergence and subsequent development of CC was virtualization. CC has enabled companies and individuals to move from local infrastructure to virtualized resources deployed in the cloud. This approach is advantageous mainly in terms of cost efficiency, where economies of scale apply. CC has also proved beneficial in the education sector, as it has opened up new opportunities for students, whether in software development or in carrying out various projects that have different hardware requirements.

In this work, we will examine the issue of monitoring a CC environment. We will try to define the motivation for such monitoring. We will also address the issue of multiple users of the platform and the way of presenting the collected data to them. We will examine the metrics that are appropriate to monitor in such an environment. Subsequently, we will try to find a specific solution that would be suitable for deploying such a monitoring system.

The motivation for working on this project is the creation of a new CC platform at KIS FRI UNIZA, whose ambition is to serve employees, students and possibly other entities, particularly in the domain of providing reliable and high-performance computing resources on demand.

The result of the work should be an experimental deployment of the selected solution. The deployment procedure should be thoroughly documented. With this deployment, we will verify the functionality of the solution, which may be deployed into a real CC system at a later stage, e.g. as part of a final thesis.

1 ANALYSIS

1.1 DESCRIPTION OF THE TOPIC

In this document, we address an experiment with monitoring a cloud running on the *OpenStack* platform, which offers mainly an IaaS service.. To begin with, it is necessary to define what specifically is important to monitor, both from the perspective of the cloud administrator and from the perspective of users. We are mainly interested in the performance of the cloud and the associated infrastructure, so that we can determine whether the provision of services is at risk. After determining the specific monitoring parameters, we will examine the monitoring options in the *OpenStack* platform, or the options offered by third parties. The aim will be to create a design of the solution.

1.2 MOTIVATION FOR CLOUD MONITORING

Cloud monitoring is of great importance both for the provider and for the end user. From the provider's perspective, monitoring enables it to manage and control both the hardware and the software infrastructure. Continuous monitoring of the cloud and its SLA (*Service Level Agreement* – agreement on the level of service) parameters provides both the provider and the customer with various kinds of information, such as the generated load, performance and quality of service (QoS – *Quality of Service*). On the basis of this information, the provider can deploy mechanisms that prevent or are able to remedy violations of the agreed SLA, whether from the customer's perspective (exceeding the contracted parameters) or from the provider's perspective (insufficient service performance). Monitoring is critical for a large number of processes in the cloud. We will discuss these processes below and evaluate the role of monitoring in each of them [1].

1.2.1 Resource planning

Before the more widespread adoption of cloud services, it was difficult for service and application developers to estimate how many computing resources they would need for the smooth operation of their product. Cloud service providers usually guarantee a certain level of QoS. This would include the performance and quantity of computing resources, which are agreed in advance when the SLA is drawn up. For developers who have their service in the cloud, this means that in a situation where they need more computing resources for the operation of their service than they anticipated, they do not need to purchase new resources; they simply need to request an increase in resources within the SLA. For the cloud provider, however, this means that it must have a monitoring system deployed so that it can track and predict the development of all the parameters involved in the process of guaranteeing QoS, for the purpose of correctly planning the expansion of the cloud infrastructure [1].

1.2.2 Resource management

Managing a complex system such as the cloud requires a deployed monitoring system that can accurately capture its state in real time. Compared with managing conventional, purely hardware-based systems, the cloud presents administrators with a new challenge. This challenge stems from one of the fundamental pillars of cloud computing, virtualization. Virtualization makes it possible to hide the high heterogeneity of hardware resources, which we regard as a benefit, but it gives rise to the need to manage both hardware and virtual resources. It must be borne in mind that virtual resources can migrate between physical machines at any time. A properly deployed monitoring system makes it possible to cope with fluctuations in available resources and rapidly changing network conditions so that all contractually agreed parameters of the services offered are

met [1].

1.2.3 SLA management

A monitoring system is the foundation and a necessity when determining compliance with SLA parameters. From the provider's perspective, monitoring makes it possible to formulate the wording of the SLA for individual clients more realistically, as well as to provide a better billing model based on knowledge of infrastructure performance [1].

1.2.4 Billing

One of the fundamental characteristics of cloud computing is the offering of so-called measured services (*measured services*). In practice, this allows the customer to pay exactly for what they have used. In the case of the IaaS service type, billing usually includes the number of virtual machines, also taking into account their specifications such as the number of vCPUs, memory size, etc. Monitoring is crucial for this process, since it provides the provider with the information needed to compile the invoice, and the customer can verify the correctness of the fees charged and track consumption [1].

1.2.5 Troubleshooting

The complex infrastructure of the cloud poses a challenge when trying to locate and resolve any problem that arises. If no monitoring system capable of identifying a problem in a specific section (e.g. on the basis of log records) were deployed in the infrastructure, numerous components (e.g. network, host) would have to be searched manually, with some of these components additionally consisting of several different layers (e.g. real and virtual hardware). A properly deployed monitoring system allows us to avoid this lengthy process and to rapidly detect and respond to errors that arise, which ultimately means shorter outages and a better service [1].

1.3 BASIC CONCEPTS FOR CLOUD MONITORING

1.3.1 Cloud layers

Based on the work of the *Cloud Security Alliance* we can divide the cloud into seven basic layers. These layers are listed and described below.

- **Facility:** this layer consists of the entire physical infrastructure, including the hardware machines hosting the computing resources as well as the network devices.
- **Network:** this layer includes the links and paths both within the cloud and between the cloud and the end user.
- **Hardware:** the individual hardware components of both computing and network devices are assigned to this layer. These may be, for example, processors, memory modules, motherboards, etc.
- **Operating system (OS):** this layer includes the software components of both the host OS (OS running on physical machines) and the user OS (OS running on virtual machines).
- **Middleware:** this layer is relevant only from the perspective of certain service types, more precisely SaaS and PaaS. It comprises the software layer between the OS and the user application.
- **Application:** this layer comprises the application that has been launched by the user of the cloud system.
- **User:** this last layer comprises the end user of the cloud system and the applications launched by them that run outside the cloud (for example, a web browser launched by the user on their own device).

From the monitoring perspective, the layers mentioned above can be understood

as the places where the individual monitoring probes should be installed [1].

1.3.2 Monitoring levels

Both high-level and low-level monitoring can be deployed in a cloud environment. For a well-functioning monitoring system, it is advisable for both levels of monitoring to be deployed. High-level monitoring deals with the state of the virtual platform itself. Low-level monitoring, on the other hand, is more concerned with the state of the physical infrastructure. For the PaaS and SaaS service types, this low-level monitoring is purely in the interest of the provider and the end user has no access to these data, whereas for IaaS both levels of monitoring are in the interest of both the customer and the provider. From the IaaS perspective, it is precisely low-level monitoring, which collects information from the various cloud layers, that is more important. Examples for the individual layers are given below:

- **Hardware layer:** diverse information on the CPU (temperature, frequency, load), memory (available, used, temperature), temperatures of other components, voltage, load of other components.
- Operating system and middleware layer: errors and software vulnerabilities.
- **Network layer:** infrastructure security by means of a firewall, an *Intrusion Detection System* (IDS) and an *Intrusion Prevetion System* (IDS).
- **Facility level:** here we could include the physical monitoring of data centres using video cameras [1].

1.3.3 Monitoring tests and collected metrics

Monitoring tests can be divided into two main categories, which we will call the network category and the compute category. We will now describe which metrics it is advisable to collect in each category.

For the compute category it is advisable to deploy tests for the following metrics.

- **Server throughput** defined as the number of requests per second
- Processor speed
- Time taken by the processor to process a single request
- Processor utilization
- **Number of I/O operations** defined as a count per second
- Memory throughput
- Response time
- Time required to start a VM
- VM start-up duration

For the network category it is advisable to deploy tests to monitor the following metrics.

- Round trip time (RTT)
- Delay variation (jitter)
- Throughput
- Loss rate
- Available bandwidth

All these metrics can be evaluated using classical statistical indicators such as the mean, median, etc. [1].

The work [2] looks at metrics in a similar way. It also divides monitoring metrics into four categories based on the layers of the cloud model which we mentioned in 1.3.1.

The individual categories, as well as the metrics that belong to them, are described below.

a) **Workload-based (*Workload-based*)** – This category relates mainly to the SaaS cloud service, since the individual metrics, which in this case are also referred to as key performance indices (*Key Performance Index* - KPI), are collected at the application layer. This category includes, for example, the response time between services, the response time in inter-process communication, the number of running threads, etc. These metrics can subsequently be merged into a single abstract metric, on the basis of which a load weight can be assigned to the individual virtual machines (VMs). This can later be translated into a QoS performance metric for the given VM [2]–[4].
b) **Compute-based (*Compute-based*)** - This category is important mainly from the perspective of the IaaS and PaaS service types. It includes metrics such as system uptime, disk throughput, CPU and RAM utilization, memory allocation and VM creation time [3], [5].
c) **Network-based (*Network-based*) –** This category may include metrics that are closely tied to the overall status of network connectivity. Specifically, these are packet counts, link throughput and the state of network interfaces [2], [3], [5].
d) **Event-based (*Events*) –** The events that are beneficial to track from the cloud monitoring perspective include, in particular, events generated by the individual cloud components at the IaaS and PaaS layer. The information from the components alone is not sufficient, which is why it is recommended to use it in correlation with the data collected from the metrics mentioned above. When used in this way, errors and deficiencies that may occur while the system is running can be identified more quickly and more accurately [2], [5], [6].

Since we are concerned mainly with monitoring a cloud platform that primarily provides the IaaS service, it is appropriate to summarize what interests us from this perspective. The resources provided by the IaaS service are primarily in the form of virtual machines. For its operation, a virtual machine needs processing compute resources and storage. It follows that, in the case of a cloud platform providing IaaS, it is advisable to deploy a monitoring solution aimed at monitoring basic metrics such as processor utilization and network load.

1.3.4 Monitoring viewpoints

A monitoring system can provide information on various aspects of the performance, behaviour and overall state of the system. Here, however, it is important to realize that how this information will be handled further depends not only on the cloud layer from which we obtained it, but also on who obtains this information next and for what purpose. In the case of an IaaS cloud service, customers are given reports on the state of their virtual machine, which includes the VM's system load, memory utilization and overall performance. The provider of such a service, however, must monitor every such VM in its infrastructure in order to be sure that it is continuously providing the service within the agreed SLA parameters. It must also monitor the physical infrastructure in order to be able to control the overall system load, the allocation of resources to new VMs or the migration of those already created. It follows that different collected data pertain to the different entities involved in the overall service, since they have different views of the given service. In general, there are two viewpoints, which are presented below [7].

- **Client viewpoint –** From this viewpoint, the cloud is perceived as an abstract entity capable of providing the required computing services. The information provided within this viewpoint should enable the client to understand the characteristics of the given service or to optimize its use. Which specific information (which metrics) will be provided depends on the wording of the SLA [7].
- **Provider viewpoint –** From this viewpoint, the cloud is a complex distributed infrastructure with numerous hardware and software elements that are interconnected in such a way as to be able to provide certain sets of services. The information provided within this viewpoint should give the operator an overview of the functioning of the individual cloud elements, their state, performance, etc. From this information, the provider should be able to check the guarantee of the SLA parameters, or it may be used to optimize system management and resource utilization [7].

For a better understanding of the viewpoints mentioned above, we will give a practical example. Let us imagine a situation in which we act as a provider of an IaaS cloud service while operating a hybrid cloud. We have built the private part of the hybrid cloud on the *OpenStack* platform, running on hardware resources owned by us. As the public part of the hybrid cloud we use Amazon EC2. As regards the monitoring system, we can view the private part from the provider viewpoint, since we can collect information from the entire infrastructure. We can no longer view the public part (Amazon EC2) from the same viewpoint, since here we act as a customer of Amazon, which provides us only with information about our VMs. We therefore view the public part from the customer viewpoint [7].

If we combine the monitoring of the individual cloud layers with the defined viewpoints, cloud monitoring can be divided into three basic types. We define these types as

1. Client-oriented monitoring
2. Virtual system monitoring
3. Physical system monitoring

For clarity, it is better to show this graphically, which is why we present Figure 1.1.

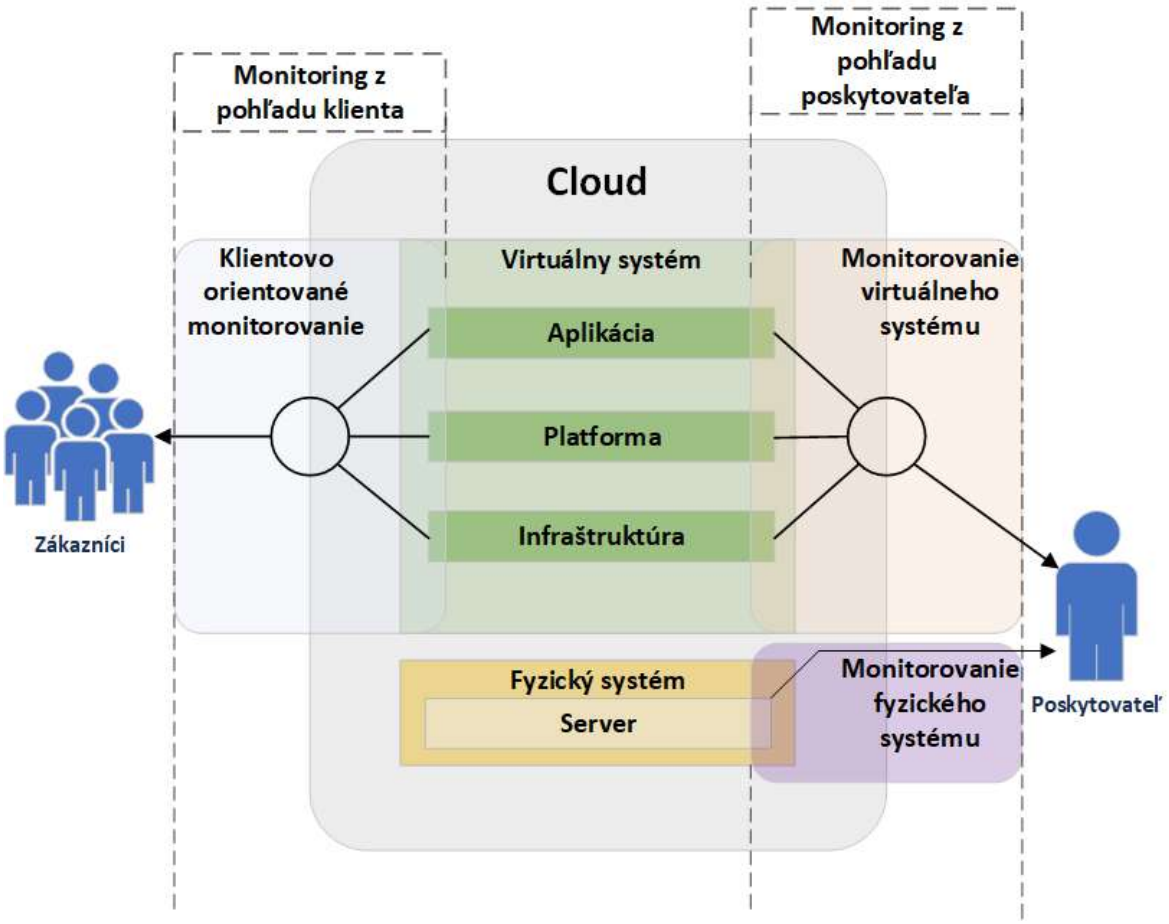


Figure 1.1 Combination of the viewpoint and level of cloud monitoring

1.4 CENTRALIZED LOG COLLECTION

This work will not deal in detail with the issue of centralized log collection. Instead, this subsection will summarize the most important parts of the diploma theses [8][9] which we consider important in the context of designing monitoring for a cloud solution.

One of the most widely used ways of recording events on various devices is by means of a solution called Syslog. It is used mainly on UNIX systems and works on the principle of the client-server model. On UNIX devices, log records are usually located in the /var/log/ directory.

When collecting larger volumes of log records, it is also necessary to address their archiving system. From our perspective, we will address optimal archiving and collection solutions in the Linux operating system, where the best solution appears to be the use of the logrotate service, whose clear documentation can be found in the Linux manual. Another part that could be of interest to us in the further solution is the ability to visualize the collected data (logs), which we would seek to achieve using an open-licence solution. Should we also need to implement a certain centralized log collection in our solution, it is precisely the works [1] and [2] that offer us 2 different solutions with which we could implement it. Since the above-mentioned works deal with this in detail, here we will only name the solutions that were used. One of the solutions is the creation of centralized log

collection using LXC containers and their subsequent visualization using the Graylog tool. Another solution presented was a collection of three open-licence products, namely Elasticsearch, Logstash and Kibana. This second solution is more interesting from our perspective, since the Kibana product (a visualization tool) also often features in connection with *OpenStack* monitoring by means of the Prometheus solution, which we would like to address later in this work.

1.5 A MONITORING SYSTEM FOR THE *OPENSTACK* CLOUD PLATFORM BUILT ON THE PROMETHEUS SOLUTION

Monitoring is an important part of cloud platforms, thanks to which we can efficiently analyse the network, manage the system, predict various events, detect errors and perform recovery operations [10].

1.5.1 Prometheus

The monitoring solution itself, called Prometheus, consists of several modules. These modules include:

- Prometheus server
  - Obtains the required data, which it then stores in a defined location
- Exporters
  - Reports data to the Prometheus server. This module runs alongside the application from which we want to obtain the required metrics.
- Pushgateway
  - Acts as a quasi-buffer in which the measured data are kept before the Prometheus server requests them.
- PromQL
  - Prometheus Query Language – retrieving data from the database.
- Alertmanager
  - Serves to inform and alert the administrator.
- WebUI
  - Serves mainly for presenting the measured data.

All these modules are interconnected and form a single functional whole [10]. The architecture of the Prometheus solution is represented in **Chyba! Nenašiel sa žiaden zdroj odkazov.**.

1.5.2 Advantages of the solution

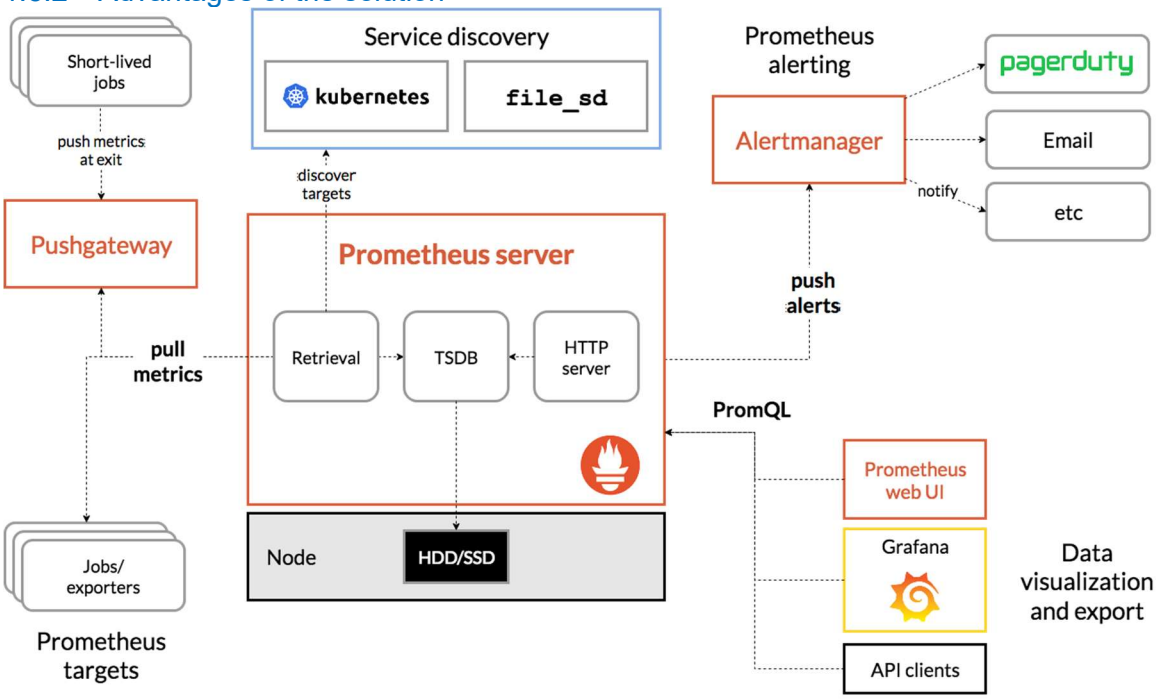


Figure 1.2 Prometheus architecture

The Prometheus solution offers several advantages. These advantages are described in more detail below.

- Provision of high-quality data
  - Prometheus stores data as time series that are precisely defined by name.
- Flexible retrieval of data from the database
  - Thanks to its own *query* language, the user can retrieve from the database exactly the data they need.
- Visualization
  - It provides several templates with which it can visualize the measured data.
- Efficient storage
  - The measured data are stored in a specific and efficient format.
- Simple operation
  - Each deployed server is independent. All binary files are statically linked and easy to deploy.
- Precise alerts
  - Thanks to the architecture of the solution, Prometheus can inform the administrator within a short time about situations that have arisen, on the basis of pre-configured alerting rules [3] [4].

1.5.3 Prometheus and SNMP

For monitoring devices based on the Simple Network Management Protocol (SNMP), the Prometheus solution can also be used. The basic prerequisite is to have a working installation of Prometheus and possibly also of the Grafana visualization tool, with Prometheus configured as a data source for Grafana. A further necessity is the presence

of an SNMP agent on the devices that we want to monitor in this way.

For Prometheus to be able to collect data via SNMP, snmp_exporter must be installed and correctly configured on a device (for example, on the device where Prometheus itself runs). This exporter allows us to collect the aforementioned data via SNMP. To configure the exporter, the configuration file, which is in yml format, can be edited manually. Manual editing of the file is, however, not recommended; instead, it is advisable to use the snmp_exporter configuration generator, which is available for download directly with the exporter itself from GitHub; more information is available at: https://github.com/prometheus/snmp_exporter/tree/main/generator [12].

1.5.4 Visualization

For data analysis and visualization, the Prometheus solution uses a tool called Grafana, which presents itself as a cross-platform tool intended for the analysis and visualization of measured data. It has several basic characteristics, which include:

- Data display
  - Grafana provides a large number of ways in which data can be visualized. The official library contains a large number of extensions from which to choose (heat maps, various types of charts such as line, pie, etc.)
- Data sources
  - The solution can draw on and visualize data from a large number of sources (databases). These sources include, for example, Graphite, OpenTSDB, Prometheus, Elasticsearch, CloudWatch, KairosDB, etc.
- Sending alerts
  - It visually displays the thresholds set for sending alerts for the most important indicators.
- Mixed charts
  - It can display several different collected metrics, from different datasets, in a single chart [10].

1.5.5 Possible deployment method

In the literature we studied, we came across a specific method of deploying the Prometheus solution in the *OpenStack* platform. In this solution, 2 compute nodes, one storage node and one controller node were deployed in the cloud. Prometheus was run on an external machine that handled monitoring. A so-called node exporter was deployed on each node, collecting and sending mainly system information about each node. In addition, an *OpenStack* exporter was deployed on the central node. It was responsible for collecting basic data from the *OpenStack* platform. The collected parameters are shown in the table below [10].

Node exporter		***OpenStack* exporter**	
Parameter name	Detailed description	Parameter name	Detailed description
ARP	Collects ARP statistics	(*Meminfo*) Memory information	Collects data on main memory
CPU	Collects statistics on processor utilization	VCPUs	Collects data on virtual processors
(*Diskstats*) Disk statistics	Collects information on I/O	RAM	Collects main memory statistics
(*Filesystem*) File system	Monitors disk utilization	Disk	Collects data such as free disk space and its utilization
(*Loadavg*) System load	Collects information on system load	Instances	Collects statistics on virtual machines, for example their number
(*Meminfo*) Memory information	Collects data on main memory	Swift	Collects data on the Swift module

We will attempt to deploy the Prometheus solution in our experimental environment in order to verify that it works and to check its functionality. The deployment will be covered in the next chapter.

2 RESULTS OF THE WORK – EXPERIMENTAL DEPLOYMENT OF PROMETHEUS + GRAFANA

Our test infrastructure is located in the department network behind a firewall. The public IP address 158.193.153.30 was assigned to us for the ingress router. In our internal network behind the ingress router, we decided to use the private IPv4 range 192.168.0.0/24. The solution will consist of implementing the ingress router and the Prometheus server. Initially, Prometheus will monitor only itself (sample configuration); to monitor our cloud, we will have to carry out the corresponding configuration. Our aim will be to monitor the deployed *OpenStack* provided at KIS.

2.1 INSTALLATION AND DEPLOYMENT OF THE INGRESS ROUTER

As the router, we decided to deploy the server distribution Ubuntu 20.04 LTS, mainly because of its ease of use, which stems from previous experience of working with this OS. We handle routing policies using nftables.

2.1.1 Installation of Prometheus

The monitoring solution consists of three separate tools (Prometheus, Grafana and *OpenStack* exporter for Prometheus). For the experimental deployment, we will install all these tools on a single VM. It is, however, important to note that installation on a single machine is not a prerequisite for functionality. Each tool can be installed on a separate system, provided that we have network connectivity between the individual devices on the specified ports. In our case, as with the router, we use the Ubuntu 20.04 LTS OS. It is important to keep this system updated to ensure correct functionality. We will deploy Prometheus first, following the procedure below.

There are many available ways to install Prometheus, such as from precompiled binary files, directly from the source code or using a Docker image. In our deployment, we used a precompiled binary file. These files, not only for Prometheus but also for its other direct extensions, are available on the website https://prometheus.io/download/. Installation is possible for a large number of operating systems (windows, linux, darwin) as well as various processor architectures. We naturally chose the linux operating system with the amd64 architecture and the LTS version 2.37.1. To download the binary file, we use the wget tool. This tool should be installed together with the Ubuntu OS, but otherwise it can be installed using `#sudo apt-get install wget.` For the download, we also need the URL link to the binary file. In our case, we download into the home folder of the logged-in user using: `#wget https://github.com/prometheus/prometheus/releases/download/v2.37.1/prometheus-2.37.1.linux-amd64.tar.gz`
We will see the download process similar to the following figure.

We can now easily verify whether the file has been downloaded using the `#ls` command. In the listing, we should see our downloaded file, similarly to the figure below.

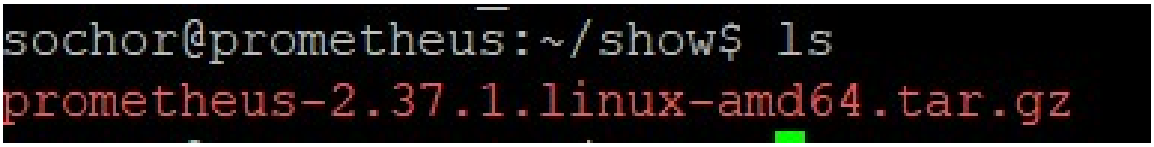


Figure 2.2 Folder show with Prometheus.tar

After downloading, we need to extract the archive so that we can proceed with the installation itself. We do this using the tar tool, specifically with the -xvzf switch. The meaning of the switch is as follows: X for extracting the archive, V for verbose output, Z because the file ends in .gz, and f for creating a folder with the same name as the archive. In our case, the command for extraction and its subsequent output, together with a listing of the contents of the current directory, looked as follows.

```
sochor@prometheus:~$ wget https://github.com/prometheus/prometheus/releases/down
load/v2.37.1/prometheus-2.37.1.linux-amd64.tar.gz
--2022-10-15 12:08:27--  https://github.com/prometheus/prometheus/releases/downl
oad/v2.37.1/prometheus-2.37.1.linux-amd64.tar.gz
Resolving github.com (github.com)... 140.82.121.3
Connecting to github.com (github.com)|140.82.121.3|:443... connected.
HTTP request sent, awaiting response... 302 Found
Location: https://objects.githubusercontent.com/github-production-release-asset-
2e65be/6838921/cc8371d5-dfed-402a-afa7-56e733c1738a?X-Amz-Algorithm=AWS4-HMAC-SH
A256&X-Amz-Credential=AKIAIWNJYAX4CSVEH53A%2F20221015%2Fus-east-1%2Fs3%2Faws4_re
quest&X-Amz-Date=20221015T120827Z&X-Amz-Expires=300&X-Amz-Signature=50169e9ac961
3829885facfc8cb86c2479723699cc9dfed64c66a5d36e4cd29f&X-Amz-SignedHeaders=host&ac
tor_id=0&key_id=0&repo_id=6838921&response-content-disposition=attachment%3B%20f
ilename%3Dprometheus-2.37.1.linux-amd64.tar.gz&response-content-type=application
%2Foctet-stream [following]
```

Figure 2.1 Prometheus download process

```
sochor@prometheus:~/show$ tar -xvzf prometheus-2.37.1.linux-amd64.tar.gz
prometheus-2.37.1.linux-amd64/
prometheus-2.37.1.linux-amd64/consoles/
prometheus-2.37.1.linux-amd64/consoles/index.html.example
prometheus-2.37.1.linux-amd64/consoles/node-cpu.html
prometheus-2.37.1.linux-amd64/consoles/node-disk.html
prometheus-2.37.1.linux-amd64/consoles/node-overview.html
prometheus-2.37.1.linux-amd64/consoles/node.html
prometheus-2.37.1.linux-amd64/consoles/prometheus-overview.html
prometheus-2.37.1.linux-amd64/consoles/prometheus.html
prometheus-2.37.1.linux-amd64/console_libraries/
prometheus-2.37.1.linux-amd64/console_libraries/menu.lib
prometheus-2.37.1.linux-amd64/console_libraries/prom.lib
prometheus-2.37.1.linux-amd64/prometheus.yml
prometheus-2.37.1.linux-amd64/LICENSE
prometheus-2.37.1.linux-amd64/NOTICE
prometheus-2.37.1.linux-amd64/prometheus
prometheus-2.37.1.linux-amd64/promtool
sochor@prometheus:~/show$ ls
prometheus-2.37.1.linux-amd64  prometheus-2.37.1.linux-amd64.tar.gz
```

Figure 2.3 Extracting the Prometheus archive

As we can see in the figure above, a new folder has appeared in the directory, containing all the files needed to run the program. When we move into the newly created folder, its contents look as shown in the following figure.

```
sochor@prometheus:~/show/prometheus-2.37.1.linux-amd64$ ls
console_libraries  consoles  LICENSE  NOTICE  prometheus  prometheus.yml  promtool
```

Figure 2.4 Contents of the Prometheus folder

We now move on to the launch itself, for which there are two different ways to approach this step. The simpler way is to run the *prometheus* executable itself using `#./prometheus`. This solution is, however, one-off, and after every system restart it will be necessary to start Prometheus manually again. For this reason, we recommend creating Prometheus as a service, thanks to which we will be able to control it using the *systemctl,* tool and also easily configure it to start at system boot.

2.1.2 Prometheus as a service

Setting up the application as a service is relatively simple and straightforward. The procedure is as follows:

1. We create a new file in the **/etc/systemd/system/** directory. The file must be named in the form **<*FILE_NAME*>.service.** In our case, we created the file using the command:
   `#touch /etc/systemd/system/prometheus.service`.
   In our case, we verified the creation of the as yet empty file as shown in the figure below.

```
sochor@prometheus:~$ cd /etc/systemd/system/
sochor@prometheus:/etc/systemd/system$ ls | grep prometheus
prometheus.service
```

Figure 2.5 Verification of the creation of the prometheus.service file

2. We now need to fill the created file with an appropriate configuration. Our sample configuration is given below. Lines beginning with the characters "#" and ";" are ignored in systemd configuration files and can therefore be used for comments, as in our case below.

```
[Unit]
#Informative description
Description=Prometheus Server

#Link to the application documentation
Documentation=https://prometheus.io/docs/introduction/overv
iew/

#Specifies that the application is started only after the
network #services have started
After=network-online.target

[Service]
#Specifies that the service is run under the root user
User=root

#In the event of failure, attempts to restart the service
Restart=on-failure

#Path to the application executable together with the
required #switches (folder for storing the collected data,
#location of the configuration file (see 2.1.4))
ExecStart=/home/sochor/prometheus-2.37.1.linux-
amd64/prometheus --
storage.tsdb.path=/var/lib/prometheus/data/ --
config.file=/home/sochor/prometheus-2.37.1.linux-
amd64/prometheus.yml

[Install]
#Defines that the service is started at system boot
WantedBy=multi-user.target
```

3. Next, the systemctl daemon needs to be restarted. We do this by entering the command
   `#sudo systemctl daemon-reload`
4. Finally, we start the prometheus service using
   `#sudo systemctl start prometheus.`
   The status of the service can easily be checked by entering
   `#sudo systemctl status prometheus.`
5. For the application to start together with the OS, this service needs to be enabled using
   `#sudo systemctl enable prometheus`.

During our attempt to create the service, we encountered a problem, which is described in more detail, together with its solution, in chapter 2.1.8. Detailed documentation on the operation and configuration of this solution is available on the website https://prometheus.io/docs/introduction/overview/.

2.1.3 Installation of the Grafana visualization tool

After installing the Prometheus solution, we recommend also deploying the Grafana visualization tool, which can display, in a customizable way, data collected either by Prometheus or from other database solutions. There are several ways to install this

tool. In our case, within the experimental deployment, we opted for the installation of a .deb file, since we are working on the Ubuntu OS. Various installation methods are available online at https://grafana.com/docs/grafana/v9.0/setup-grafana/installation/debian/. Two different editions are available: OSS and Enterprise. Both can be used free of charge, with the enterprise edition being the one recommended by the developers. OSS is under the AGPLv3 licence, whereas enterprise is under the Grafana Labs License. The website states that the enterprise edition contains all the functionality of the OSS edition, but can be activated to the "full Enterprise feature set", which also includes support for enterprise plugins. Since we are considering deployment in an academic setting, we chose the OSS edition. At the time of deployment, the latest stable version of the application was 9.2.0, which we installed.

Instructions for downloading various types of files for specific OSs are available at https://grafana.com/grafana/download/9.2.0?edition=oss&platform=linux. In our case, we proceeded as follows:

```
#sudo apt-get install -y adduser libfontconfig1
#wget https://dl.grafana.com/oss/release/grafana_9.2.0_amd64.deb
#sudo dpkg -i grafana_9.2.0_amd64.deb
```

With this installation method, grafana was automatically created as a service. To start the application, we enter the following sequence of commands, expecting the status active (running):

```
#sudo systemctl daemon-reload
#sudo systemctl start grafana-server
#sudo systemctl status grafana-server
● grafana-server.service - Grafana instance
     Loaded: loaded (/lib/systemd/system/grafana-server.service; enabled;
vendo>
     Active: active (running) since Thu 2022-10-13 09:15:21 UTC; 6 days
ago
       Docs: http://docs.grafana.org
   Main PID: 886 (grafana-server)
      Tasks: 18 (limit: 19103)
     Memory: 132.4M
```

As with the Prometheus solution, it is advisable here too to enable the service to start together with the OS.

```
#sudo systemctl enable grafana-server.service
```

2.1.4 Setting up and accessing Prometheus

After a successful installation, we can verify functionality by accessing the graphical interface and set up a test configuration.

As part of the initial configuration, we will set Prometheus to monitor itself. This step is also given in the official documentation and, although it is not very useful, it will give us a basic overview of whether the solution is working correctly. The configuration file should be named prometheus.yml and saved in an accessible folder. We created this file directly in the folder containing the prometheus executable. The sample configuration for setting up self-monitoring looks as follows:

```
global:
scrape_interval:     15s # Request new data every 15s - this is the
# default value

# These labels will be attached to time series and alerts when
```

```
# communicating with external systems (remote storage, Alermanager)
  # external systems (federation, remote storage, Alertmanager).
  external_labels:
    monitor: 'codelab-monitor'

# This section lists the endpoints intended for monitoring.
# In this case, it contains only Prometheus itself.
scrape_configs:
  # Label setting for this job
  - job_name: 'prometheus'

    # Overrides the default time interval for data collection
    scrape_interval: 5s

    static_configs:
      - targets: ['localhost:9090']
```

After creating the configuration file, we need to restart the application or use a switch to specify which configuration file is to be used. If we have already created Prometheus as a service, we check the `-config.file` switch in the ExecStart section of the /etc/systemd/system/prometheus.service file. The current path to the configuration file must be given here. We then simply restart the service with `#sudo systemctl restart prometheus.service`.

After performing the restart, it is advisable to wait a few minutes until a certain amount of data has been collected. We will now describe access to the graphical interface of the application.

As the first step, we recommend checking the status of the prometheus service to verify that no problem has arisen during operation. We expect the status active (running). With the current, factory configuration, the graphical interface is accessible using http on port 9090. In our experimental environment, it was necessary to configure forwarding of this port to achieve the desired result. Access the web interface at http://[IP_ADDRESS]:9090, where the built-in expression browser is displayed, as we can see in the figure below.

This browser is used to search the collected data using an expression language, which is available online at https://prometheus.io/docs/prometheus/latest/querying/basics/. It is also advisable to verify whether Prometheus is collecting data about itself, and at the same time we will demonstrate the functionality of the browser here. One of the metrics that Prometheus collects about itself is called

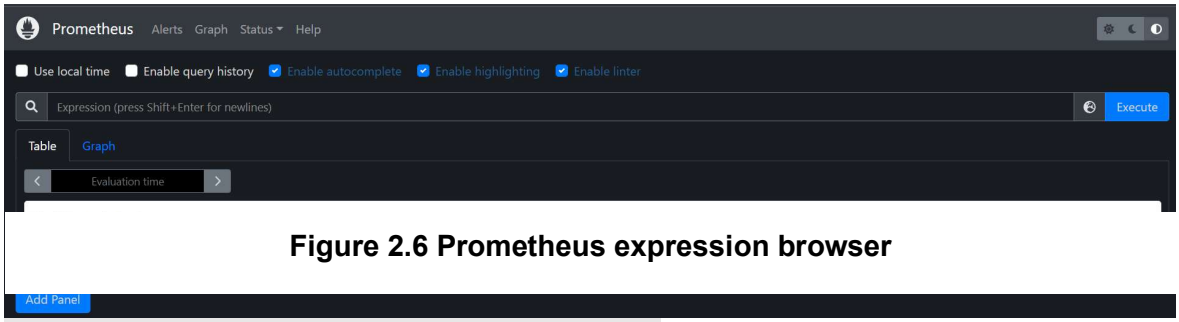


Figure 2.6 Prometheus expression browser

`prometheus_target_interval_length_seconds`, which represents the time between "target scrapes", which we could render in Slovak as the act of obtaining data from the target host. We enter the name of the metric in the designated field and execute a query for the data from the database, as shown in the next figure.

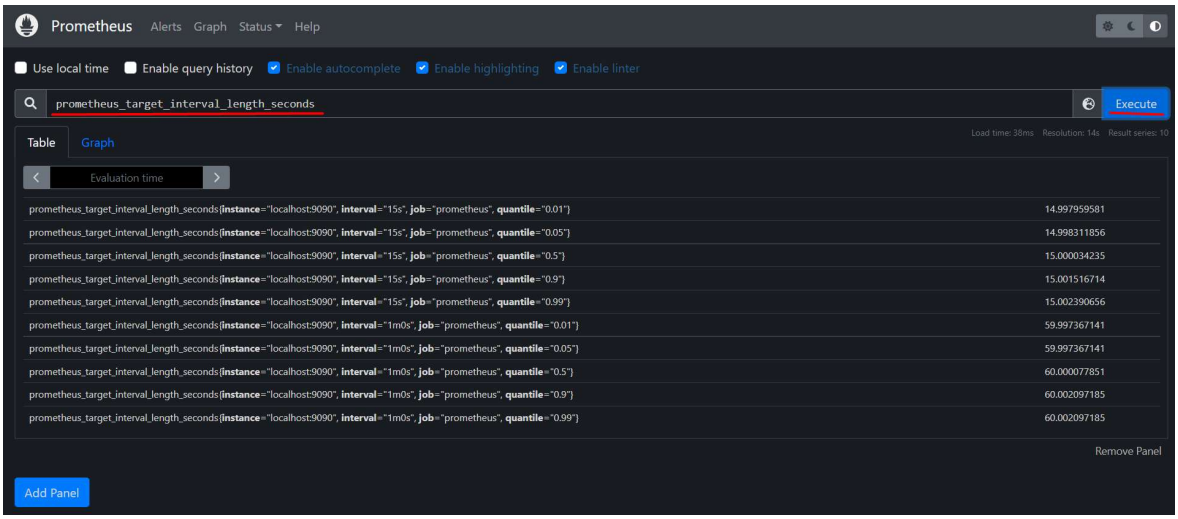


Figure 2.7 Initial data search

The result of the query is a set of time series for the metric with the specified name, but with different labels (*labels*). To demonstrate working with the browser, we will also show how we would obtain the number of results returned by the query. We use the expression `count(prometheus_target_interval_length_seconds)`; its output is shown in the following figure.

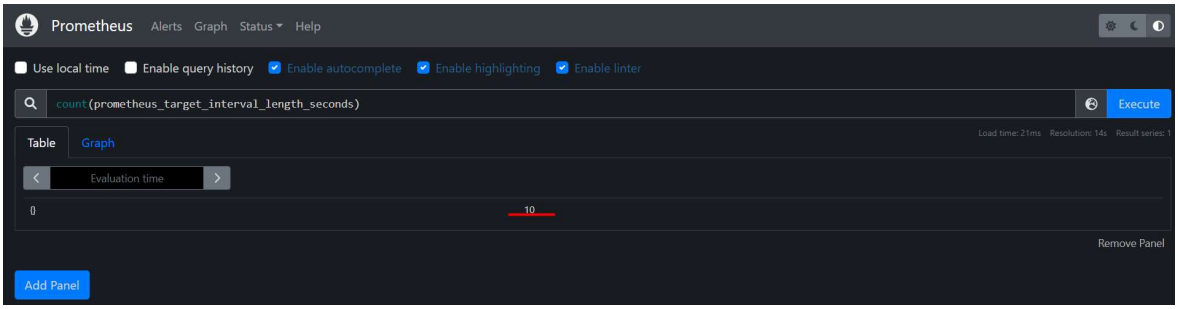


Figure 2.8 Returning the number of responses obtained for the query

The web interface also offers a certain form of graphical visualization of the collected time series; however, the official documentation states that it is advisable to use other solutions for visualization, such as Grafana.

2.1.5 Setting up and accessing Grafana

Unlike the Prometheus solution, for this visualization tool we do not currently need to perform any additional configuration to ensure basic functionality. The factory default port on which the web service runs is http/3000. We will now attempt to access the web interface of the application. We use http://[IP_ADDRESS]:3000/. We expect the login screen to be displayed, as shown below.

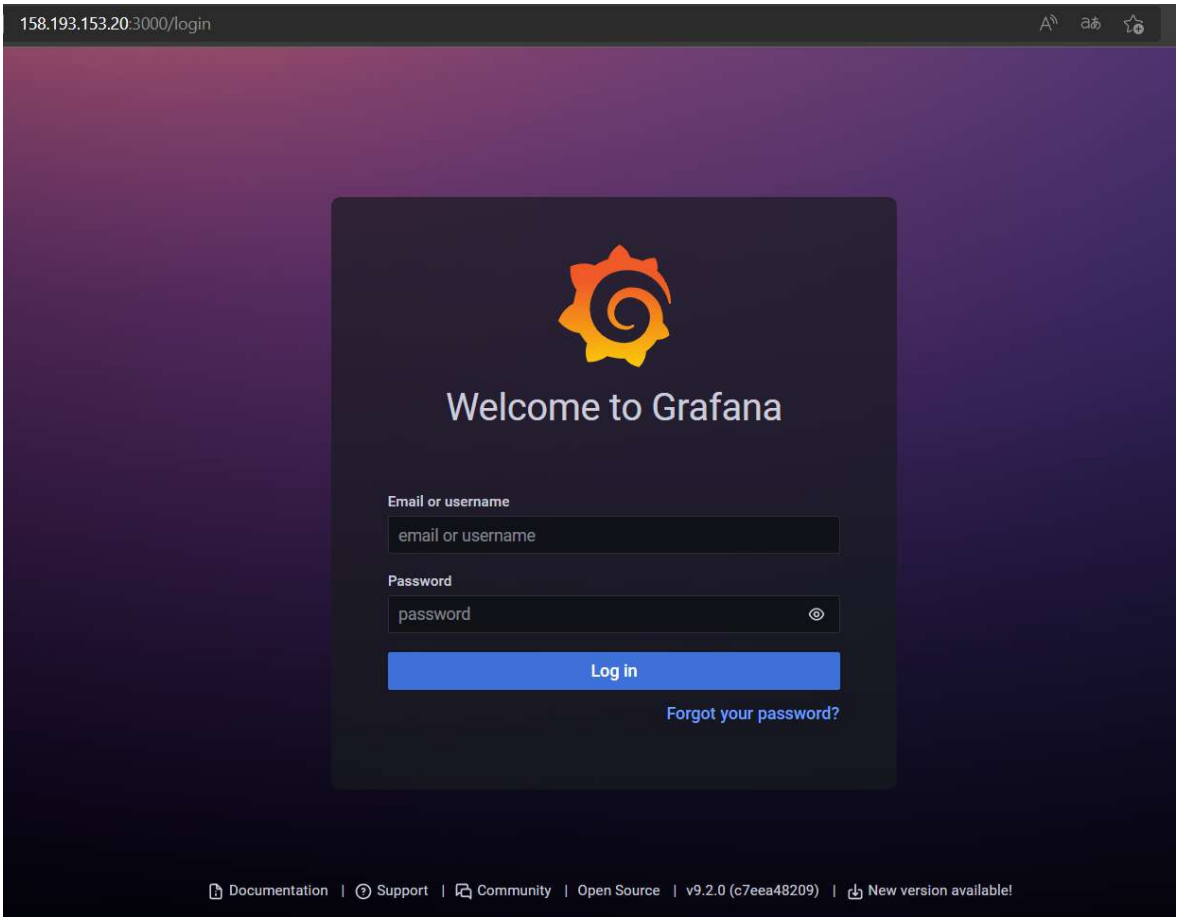


Figure 2.9 Grafana login screen

The factory default administrator account has the following credentials:

- Name: **admin**
- Password: **admin**

After successful authentication, the application prompts us to change the default password, which is a highly recommended step. We then reach the home screen. At the moment, we do not have any data visualization screen created; we will address this only after successfully collecting data from *OpenStack*.

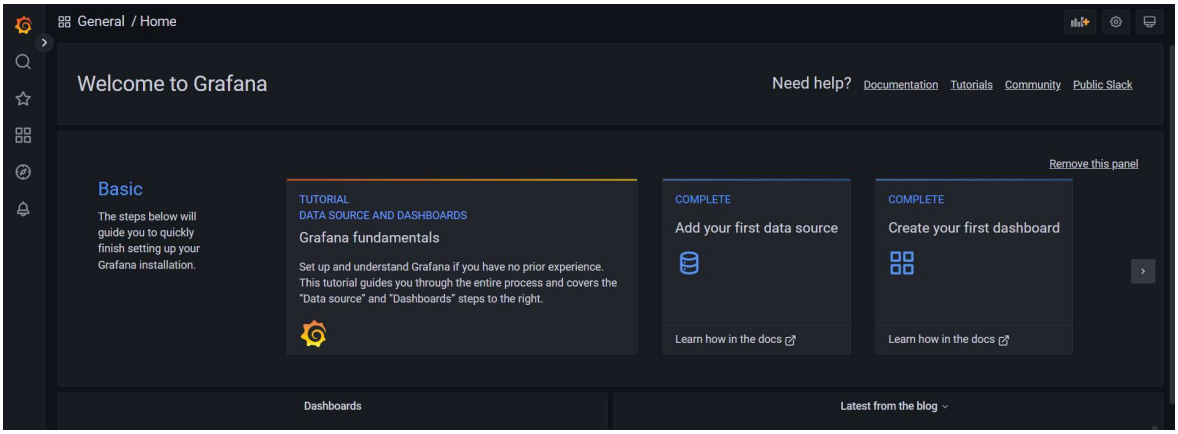


Figure 2.10 Grafana home screen

2.1.6 Deployment of the *OpenStack* exporter

A major advantage of the Prometheus monitoring solution is also the wide range of extensions that save us time when deploying and configuring the monitoring of individual parts. For *OpenStack* there is the *OpenStack exporter for Prometheus* extension, available online at [13]. The basic task of this extension is to extract data on the state of the internal components of *OpenStack* and make them available in a format compatible with the *Prometheus* solution. It is worth mentioning that the tool works by means of the RESP API, through which it communicates with the individual modules. The exporter can therefore be run on any device, provided that this device has the necessary connectivity both to the monitored cloud and to the device running Prometheus.

For the deployment, the developers provide four different installation methods:

- Using kolla-ansible
- Using helm charts
- Using a Docker image from the repository
- Using snapd

All the installation methods mentioned above are described, together with further external links (kolla-ansible, helm charts), in the git repository [13]. In our experimental deployment, we installed the exporter on the same device as Prometheus and Grafana. From the perspective of performance and scalability, this is not optimal, and we recommend deploying these tools on separate computing entities; however, in our case we are mainly concerned with verifying the functionality of the individual components, as well as of the whole, in the shortest possible time, which is why we chose the approach of installing on a single device.

For simplicity, we will install the *OpenStack* exporter using snapd, and we will also

make this tool a service to ensure that it starts together with the system boot.

We begin the installation by installing the snap using:

```
#sudo snap install --channel stable golang-openstack-exporter
```

This exporter can work in two different modes, *legacy* (a single targeted cloud) and *multi cloud* (multiple targeted clouds). For our deployment, we use *legacy* mode, since our aim is to monitor a single cloud entity.

To control the application from the console, we start with the command `#golang-openstack-exporter.openstack-exporter.` For example, to list all available switches and options, we can use the –help switch, as shown below:

```
sochor@prometheus:~$ golang-openstack-exporter.openstack-exporter --help
usage: openstack-exporter [<flags>] [<cloud>]

Flags:
  -h, --help                     Show context-sensitive help (also try
                                 --help-long and --help-man).
      --log.level="info"         Log level: [debug, info, warn, error,
fatal]
```

From the solution's perspective, controlling the application from the console is unsuitable for us, since it does not provide the option of automatic start-up after a system restart. For this reason, we need to create this snap as a service, where we will also describe the important parts of the exporter that we will configure.

The first step is to create the file **[NAME].service** in the `/etc/systemd/system` directory. Next, we need to find out the location of the executable for our snap. For this purpose, we used the procedure shown below.

```
sochor@prometheus:/etc/systemd/system$ which golang-openstack-
exporter.openstack-exporter
/snap/bin/golang-openstack-exporter.openstack-exporter
```

From the output above, it is clear that the path to the executable is /snap/bin/golang-openstack-exporter.openstack-exporter.

Of all the switches offered by the application, for the time being we will use only one mandatory one (`--os-client-config`), namely the path to the configuration file `clouds.yaml.` First, however, we need to create this file. We found experimentally that it is best to create this file in the **/var** directory. In other cases (the user's home directory and the /etc directory), the application had a problem accessing the file, which was blocked by the *apparmour* application, which is part of the *snapd* package. We present our recommended procedure as a combination of generally described steps with the attached commands that we used during deployment:

1. Creating the clouds.yaml file in the /var/[OPTIONAL_NAME] directory.
   ```
   #mkdir /var/prometheusProject
   #touch /var/prometheusProject/clouds.yaml
   ```
2. Fill the created file with the correct configuration. More information on the syntax is available online at https://docs.openstack.org/python-openstackclient/pike/configuration/index.html. In this case, we will not present our version of the completed file here, as it contains non-public information. We will only mention that we named our cloud default.

We now have all the necessary components available to create openstack-exporter as a service. Below we give a sample configuration of the file, in our case node-

exporter.service.

```
[Unit]
Description=Openstack exporter
After=network-online.target

[Service]
User=root
Restart=on-failure

ExecStart=/snap/bin/golang-openstack-exporter.openstack-exporter
--os-client-config=/var/prometheusProject/clouds.yaml default

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

We perform the necessary reload of the systemctl daemon.

```
#sudo systemctl daemon-reload
```

We start our service and check its status.

```
#sudo systemctl start node-exporter.service
#sudo systemctl status node-exporter.service
● node-exporter.service - Node exporter
     Loaded: loaded (/etc/systemd/system/node-exporter.service; enabled;
vendor preset: enabled)
     Active: active (running) since Tue 2022-11-01 11:20:33 UTC; 24s ago
   Main PID: 54524 (openstack-expor)
      Tasks: 0 (limit: 19103)
     Memory: 812.0K
     CGroup: /system.slice/node-exporter.service
             ▸ 54524 /snap/golang-openstack-exporter/97/bin/openstack-
exporter --os-client-config=/var/prometheusProject/clouds.yaml default
```

We verify functionality by accessing the exposed metrics. Openstack-exporter collects data directly from OpenStack and makes them available on port 9180 on the host device. We therefore try to access this device on the specific port and expect a text representation of the collected metrics to be displayed.

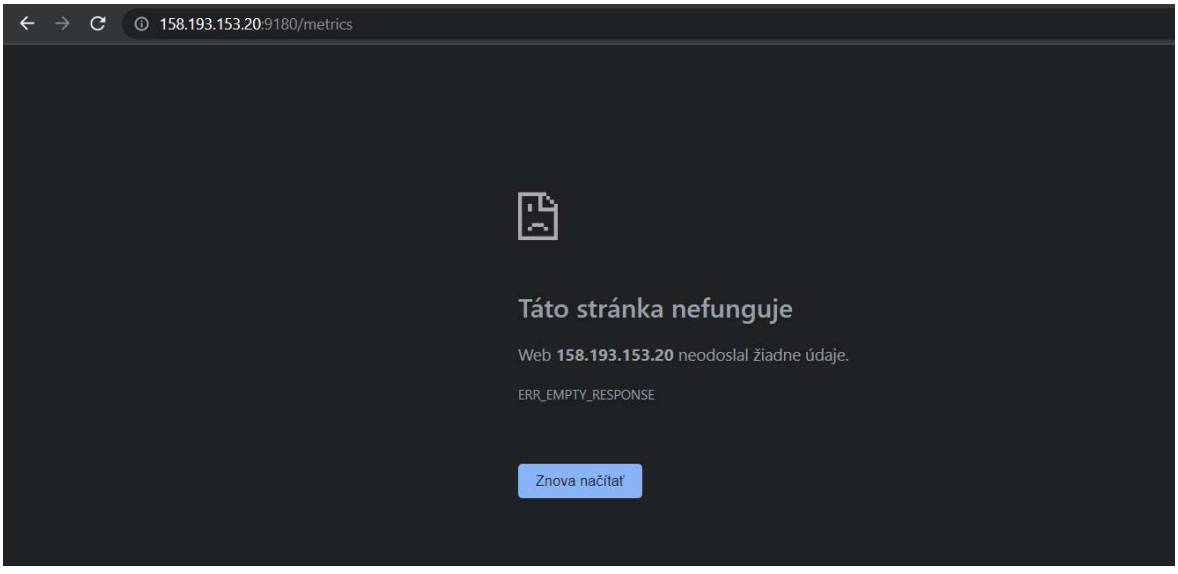


Figure 2.11 Initial test of openstack-exporter

As we can see in the figure above, we did not achieve the expected result.

2.1.7 Setting up a new source for Prometheus

We configure Prometheus to collect data from the above-mentioned exporter as well. There we will also be able to verify the state of this service. The procedure is straightforward: we add a new job to the Prometheus configuration file (prometheus.yml). In the configuration file, in the *scrape_configs:* section, we add the following configuration:

```
scrape_configs:
  - job_name: "openstack_exporter_metrics"
    scrape_interval: 60s
    scrape_timeout: 60s
    metrics_path: "/metrics"
    static_configs:
      - targets: ["localhost:9180"]
```

Prometheus can reload configuration changes even while running; this is described, together with the procedure, in the official documentation, but for simplicity we only restarted the service using `#sudo systemctl restart prometheus.service`. Now, when we access the Prometheus graphical interface, we can verify the status of the newly configured exporter in the STATUS->TARGETS section (see Figure 2.12).

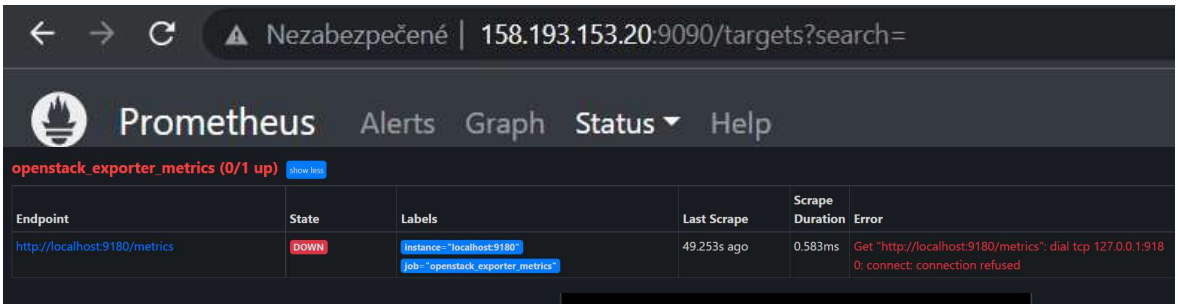


Figure 2.13 Initial state of openstack-exporter

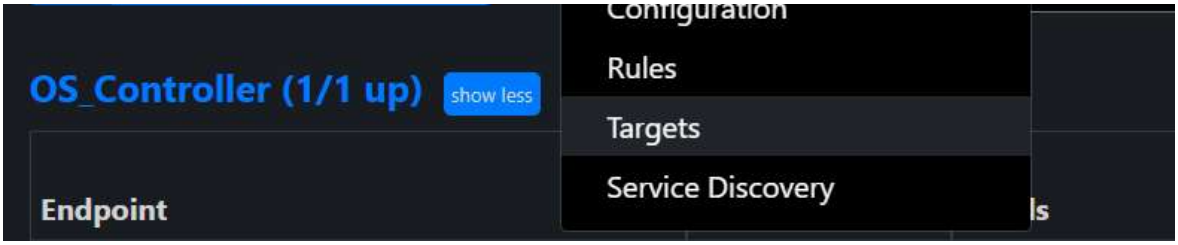


Figure 2.12 Prometheus status->targets

As Figure 2.13 shows, the service status is down. We resolved this problem experimentally, and the procedure with a description is given below.

Even though the service status indicates active (running), we found experimentally that after a certain number of accesses to the given port the service stops running. We can even say that the service was not running correctly at all. Through investigation we found that the error lies in the path to the executable file. Let us list the status of the

service once more.

```
#sudo systemctl status node-exporter.service
● node-exporter.service - Node exporter
     Loaded: loaded (/etc/systemd/system/node-exporter.service; enabled;
vendor preset: enabled)
     Active: active (running) since Tue 2022-11-01 11:20:33 UTC; 24s ago
   Main PID: 54524 (openstack-expor)
      Tasks: 0 (limit: 19103)
     Memory: 812.0K
     CGroup: /system.slice/node-exporter.service
             ▸ 54524 /snap/golang-openstack-exporter/97/bin/openstack-
exporter --os-client-config=/var/prometheusProject/clouds.yaml default
```

At the end of the output above, we can notice that a process with PID 54524 was started, which runs by executing the file at the given path with the given switches. However, this does not match the path that we specified in the service configuration. We found experimentally that if we specify in the service configuration the complete path provided by the service status output, the exporter will work correctly. Therefore, in our case, in the file /etc/systemd/system/node-exporter.service we change the path at *ExecStart.* In our case, the modified line looks as follows: `ExecStart=/snap/golang-openstack-exporter/97/bin/openstack-exporter --os-client-config=/var/prometheusProject/clouds.yaml default.`

- We need to restart the systemctl daemon again. `#sudo systemctl daemon-reload`
- Now we start the service and list its status

```
#sudo systemctl status node-exporter.service
● node-exporter.service - Node exporter
Loaded: loaded (/etc/systemd/system/node-exporter.service; enabled;
vendor preset: enabled)
     Active: active (running) since Tue 2022-11-01 11:31:56 UTC; 33s ago
   Main PID: 55504 (openstack-expor)
      Tasks: 11 (limit: 19103)
     Memory: 20.4M
     CGroup: /system.slice/node-exporter.service
             └─55504 /snap/golang-openstack-exporter/97/bin/openstack-
exporter --os-client-config=/var/prometheusProject/clouds.yaml default
```

- We verify the target status in the Prometheus graphical interface. (optionally, we can also access the exporter metrics on port 9180)

2.1.8 Connecting Grafana+Prometheus and deploying the panel for openstack-exporter

A major advantage of the Grafana solution is the availability of a huge number of pre-built templates. Thanks to this, we can save a great deal of configuration and setup when manually creating screens (dashboards).

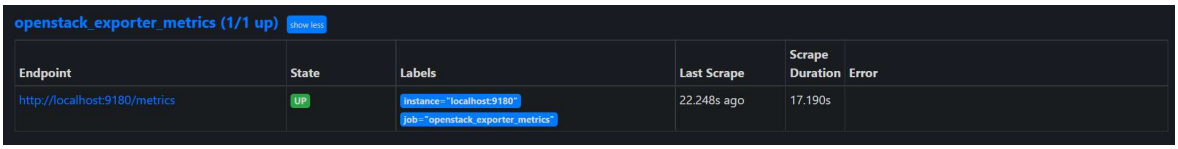


Figure 2.14 Functional state of openstack-exporter

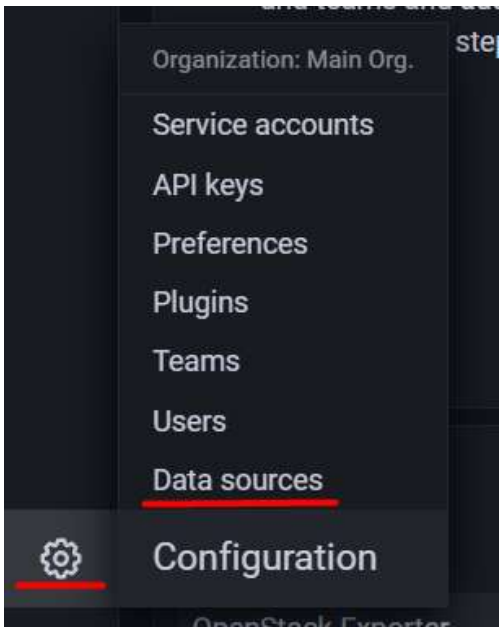


Figure 2.15 Grafana Data sources

First, Grafana needs to be configured so that it can connect to our Prometheus. After logging in to Grafana (see 2.1.5), we select the gear icon on the left panel and the option "*Data sources*". This step is shown in the figure on the right. A new menu then opens, where we select the blue "*Add data source*" button. A list of a large number of supported sources is displayed. We select the Prometheus source, which in our case was listed first. After clicking the option, a new source is created and a configuration screen is displayed. Due to its size, we will not include it here as a figure. However, it is important to configure all the necessary fields. In our case, for the test deployment, we configured only the name and URL as follows.

- Name: Prometheus_first test
- URL: http://localhost:9090

At the very bottom of the configuration screen, we then find the "*Save & test*" button. This tests whether Grafana has access to the configured source. We expect the message "*Data source is working*". With this, we have successfully configured our new source for obtaining data for visualization. We therefore move on to setting up the monitoring dashboard for our OpenStack exporter module.

For the monitoring dashboard, we will use a pre-built template configured by the author of this module himself. Grafana offers a large number of ready-made templates for monitoring dashboards on its website. We start by opening the website https://grafana.com/grafana/dashboards/14854-openstack-exporter/. Two options are available: downloading the JSON configuration file or, more simply, copying the ID. We choose the ID option because it is simpler.

After copying the ID, we must navigate in Grafana, in the side panel, to the Dashboards->import menu, as shown in Figure 2.16.

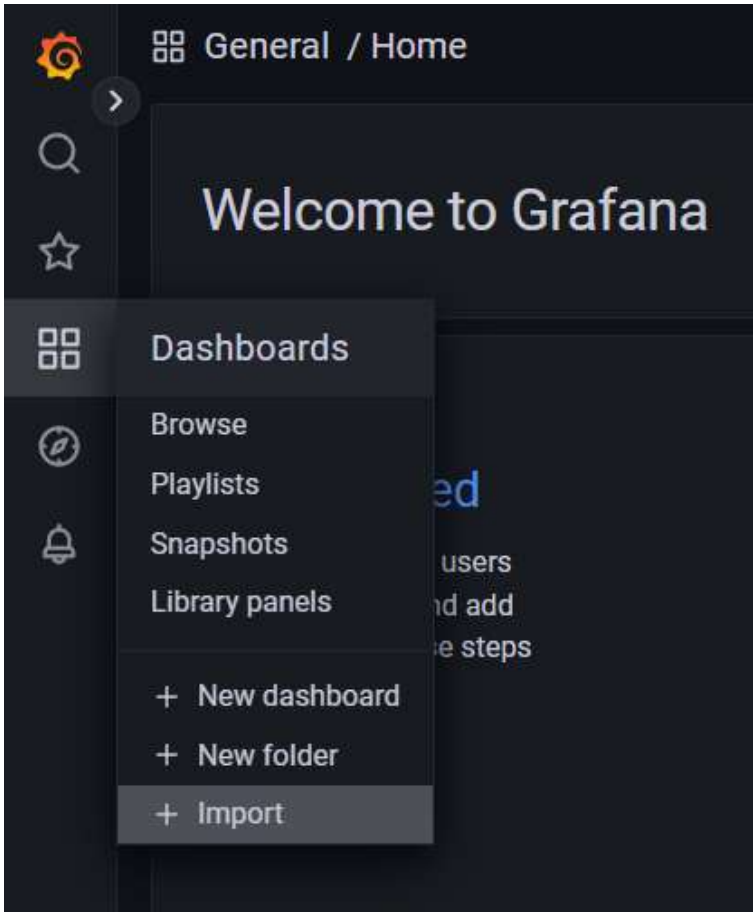


Figure 2.16 Importing the monitoring dashboard p1

Figure 2.16 Importing the monitoring dashboard p1

Next, we paste the copied ID into the designated field (import via grafana.com) and click Load. Please note that in this step we assume that the server on which Grafana is deployed has internet access. Without such access, it is necessary to use the method based on the JSON configuration file.

An import screen appears in which we can change and specify parameters such as the dashboard name, folder, UID and, most importantly, the source from which data are to be loaded into the dashboard. In our case, this will be the configured Prometheus source. Below we also provide a figure as an example.

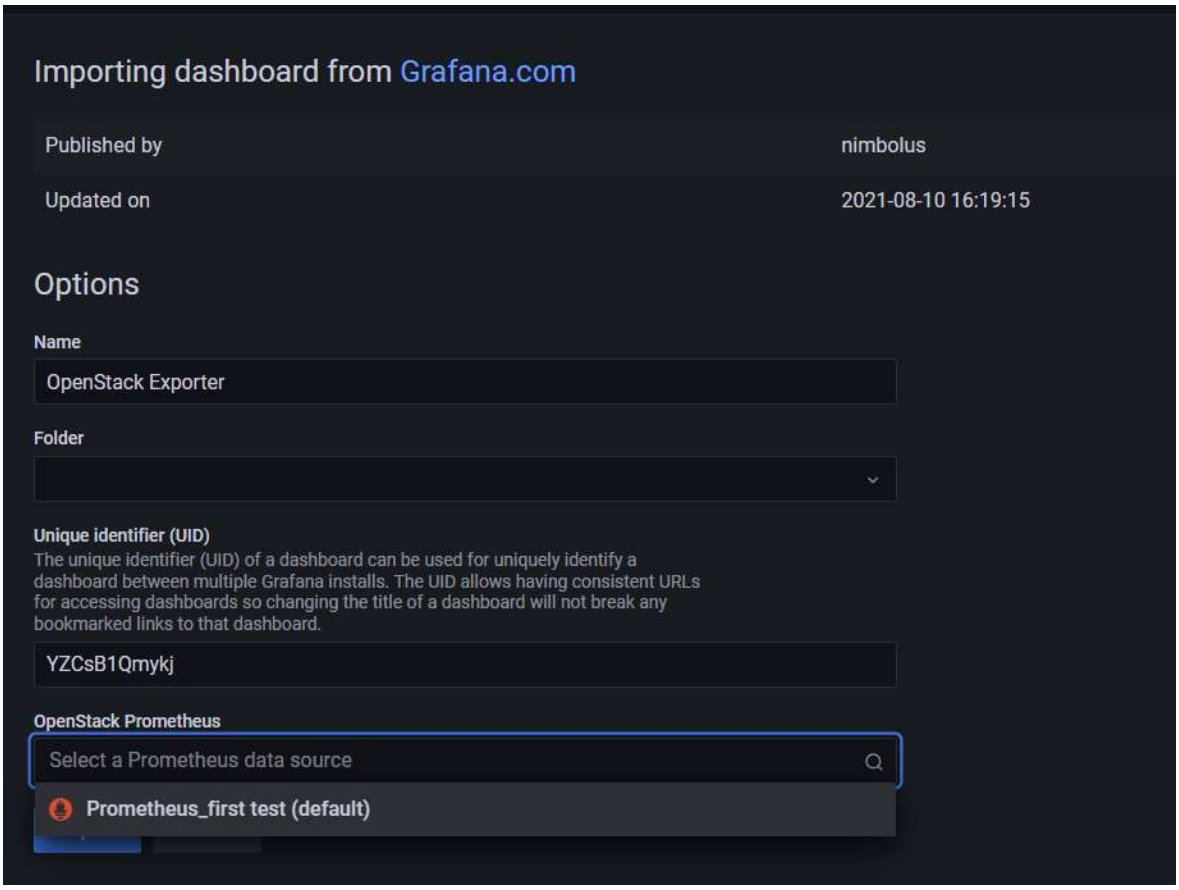


Figure 2.18 Import screen for the dashboard

We click Import and the process is complete. If we have set everything up correctly, we should have a new dashboard in the menu with information about our OpenStack platform, similar to Figure 2.17.

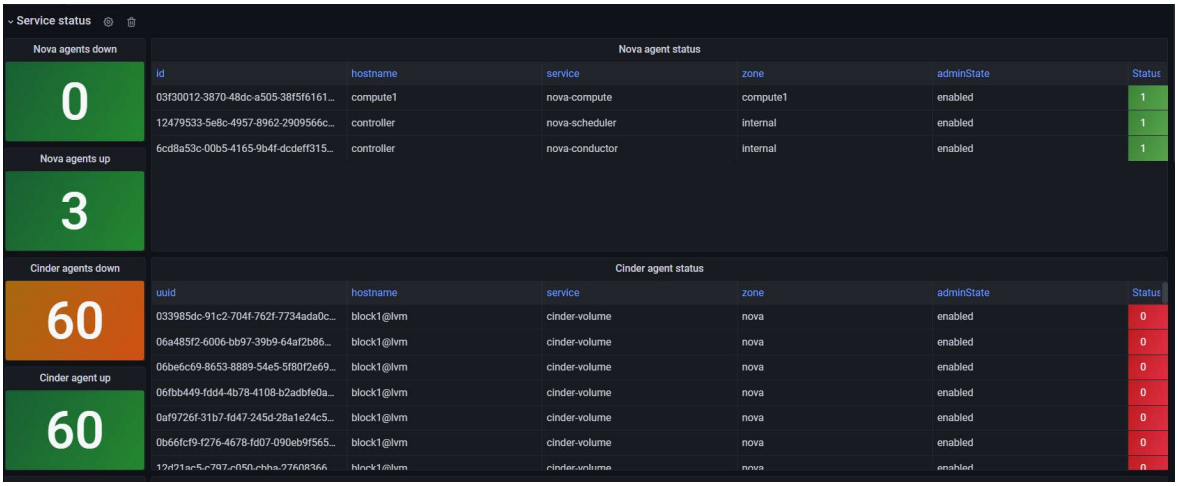


Figure 2.17 OpenStack exporter dashboard

3 DEPLOYMENT OF THE FUNCTIONAL SOLUTION

For the functional solution, we envisage not only monitoring the OpenStack platform itself but also monitoring the adjacent physical infrastructure. It is also appropriate to design a solution for monitoring the VMs themselves in the CC platform, not only the platform as a whole.

We start with the proposed topology of the solution. We will work with one controller node and one compute node, which are within a single separate network. Furthermore, we have one server available, which we configured in the previous chapter and on which Prometheus is deployed together with Grafana. The topology is shown in the figure below.

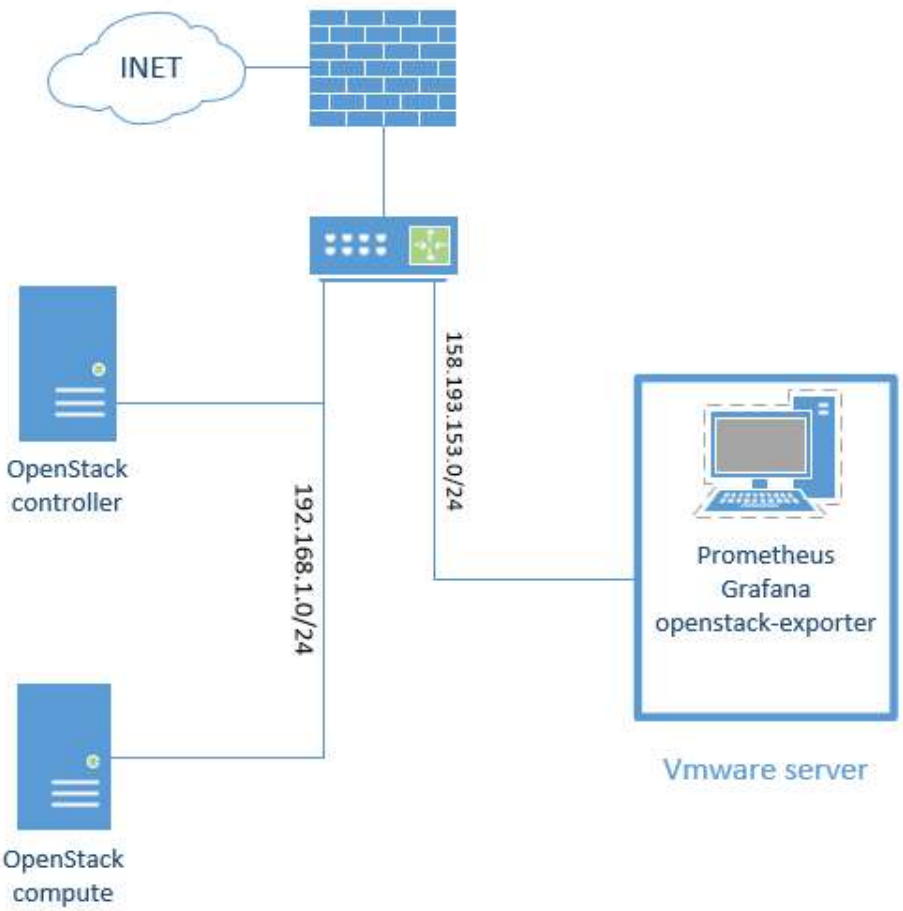


Figure 3.1 Topology of the functional deployment

The topology above shows that our monitoring server is located in a different network from the OpenStack nodes. There is a firewall between these networks. It is important to note that, for correct functionality, these networks must have connectivity between them, with at least the http service allowed, and subsequently various ports as required by the Prometheus technology.

3.1 REQUIREMENTS AND SYSTEM ELEMENTS

We require the system to monitor:

1. **State of the physical infrastructure –** we will do this using a plugin for Prometheus called **node exporter.** This plugin works on Linux systems and exposes a wide range of hardware and kernel metrics from the host system. By default, it exposes metrics on port 9100, but this can be reconfigured as needed.
2. **State of the OpenStack platform –** we also address this using Prometheus. The deployment is described in the previous chapter, since we used this deployment to demonstrate the Prometheus + Grafana configuration.
3. **State of virtual machines in the CC platform –** for this solution, we envisage using the *collectd* tool, together with the *virt* and *write_prometheus* plugins. Using these plugins, we will be able to collect metrics from the hypervisor on the compute node and then expose them to Prometheus for collection.

3.2 MONITORING THE STATE OF THE PHYSICAL INFRASTRUCTURE

Monitoring the physical infrastructure is common practice for an operator of any type of equipment. Naturally, the operator is motivated to know the state of its systems. In our case, we will use Prometheus to monitor the nodes, together with the plugin **node-exporter.** This plugin will collect a large amount of information about the operation and state of the given system and expose it via http on port 9100 (this can be changed). We then need to configure a new source in Prometheus so that it stores these collected data for visualization.

In our case, we install the mentioned plugins on both CC nodes. The installation procedure is very well documented in the official documentation, so we will not repeat it here. It is available at: https://prometheus.io/docs/guides/node-exporter/. After installation, a new data source must be added to Prometheus. We have already documented this procedure in 2.1.7, so we will not document it in detail again. For simplicity, we can use a detailed, preconfigured Grafana template to visualize the data, which is available at https://grafana.com/grafana/dashboards/1860-node-exporter-full/. As in the previous case, we documented the procedure for adding a monitoring dashboard in 2.1.8. The same procedure must be followed. The result should be a monitoring dashboard as shown in the figure below.



Figure 3.2 Node-Exporterr monitoring dashboard

After this step, we are collecting, and can also evaluate, metrics from the physical nodes. The platform also provides a tool for alerting administrators if a metric exceeds a predefined value. However, we did not have time to address this feature within the project; nevertheless, it is a very important aspect of a monitoring system, so this functionality is necessary in a production deployment.

3.3 MONITORING THE STATE OF VMS IN THE CC PLATFORM

The system administrators' requirement was to also have an overview of what is happening inside the platform and of the state of the VMs within it. The motivation is, for example, to identify redundant machines that are no longer used, or to detect those that unnecessarily use too many resources, which could raise suspicion that their users are using them for an activity other than that for which the VM was provided.

To solve this problem, after examining the available solutions, we decided to use the system statistics collection daemon **collectd.** We will use this daemon together with the following plugins:

- **Virt –** This plugin uses the *libvirt* virtualization API, which was developed by RedHat. Its task is to collect statistics about virtualized hosts in the system. It can be used to collect the utilization of the CPU, network interfaces and block devices (storage), without the need to install any software on these host systems. Data are collected directly from the hypervisor. Xen, Qemu and KVM back-ends are supported [14].
- **Write Prometheus –** The plugin creates a web server on port 9103 (configurable) and accepts requests from Prometheus to collect data from this endpoint [15].

3.3.1 Deployment of the solution

Since VMs run only on the compute node, we will install the collectd daemon only on that system. Again, there are several ways in which installation is possible. The node runs the CentOS7 operating system, for which an installation package is available, so in our case we chose this option. All other installation options, as well as the complete documentation, are available online at https://collectd.org/wiki/index.php/Main_Page.

During our deployment, we did not avoid a larger number of problems. Some originated from the OS itself and our limited knowledge of working with it. For the future, we recommend a Debian/Ubuntu-based OS for deployment. Our installation procedure, including the problems, is described below. To be precise, we performed the entire installation as the root superuser.

```
#installing collectd; first we need to enable the epel repository
yum install epel-release
```

In our case, a DNS problem arose, where yum could not find the necessary mirrors. The solution was to modify the resolv.conf file, in which we added another DNS server (8.8.8.8). If the installation is successful, we proceed to the installation of the daemon itself.

```
#Expected command
yum install collectd
```

After entering this command, we encountered the message “No package found”. After investigation, we found a solution. It is necessary to specify in the command that the epel repository should be used.

```
#Installing collectd using the epel repository
yum --enablerepo=epel install collectd
#Verifying the installation
systemctl status collectd

collectd.service - Collectd statistics daemon
   Loaded: loaded (/usr/lib/systemd/system/collectd.service; disabled;
vendor preset: disabled)
   Active: active (running) since Sun 2023-01-29 15:12:55 CET; 3 days ago
     Docs: man:collectd(1)
           man:collectd.conf(5)
```

Now we have the daemon installed, and its configuration needs to be changed. Based on that configuration, it activates the requested plugins itself. It is important to mention here that some plugins depend on various libraries and packages (this is always stated in the collectd plugin documentation). If these are not installed, they must be installed beforehand for correct functionality. Below we present the blocks of the extensive configuration file (*/etc/collectd.conf* in our case) that we modified so that the daemon activates the required plugins and collects data from the hypervisor.

In the LoadPlugin section, the following lines need to be uncommented (remove the # character).

```
#LoadPlugin virt
#LoadPlugin write_prometheusň
```

In the Plugin Configuration section, we first modify the configuration of the virt plugin. In our case, it looks as follows:

```
<Plugin virt>
        Connection "qemu:///system"
        RefreshInterval 60
        BlockDeviceFormat "target"
        HostnameFormat "uuid"
        InterfaceFormat "address"
        PluginInstanceFormat "uuid"
        ExtraStats "cpu_util disk_err domain_state job_stats_background
perf vcpupin"
</Plugin>
```

An explanation of the meaning of the individual lines is available online at: https://collectd.org/documentation/manpages/collectd.conf.5.shtml#plugin_virt.

Now we modify the configuration of the Write Prometheus plugin. In our case, we only uncommented the configuration; we had no reason to change the default port.

```
<Plugin write_prometheus>
        Port "9103"
</Plugin>
```

To activate the plugins, we now need to restart the daemon by entering the following command.

```
systemctl restart collectd
```

We expect the status active (running). In our case, we observe the error message shown below.

```
compute1 collectd[24579]: virt plugin: Array index out of bounds:
tag_index = 10
```

However, the daemon works as expected, and so far we have not been able to identify either the cause of the problem or whether it has any effect on the operation of the daemon.

The last step is to add another source to Prometheus, as already done several times before in this work. For this plugin, there is no suitable preconfigured template for visualizing the collected metrics, so we will have to create the monitoring dashboard manually. However, due to lack of time, we will not address this in this work.

CONCLUSION

Within this experiment, we managed to explore the topic of monitoring a CC platform built primarily on *OpenStack* technology. We defined the motivation for monitoring and examined metrics suitable for tracking. Furthermore, we attempted to link the topic of the service level agreement with the monitoring itself. We described the selected solution *Prometheus* together with the visualization platform *Grafana*.

In the practical part of the work, we managed to deploy the examined *Prometheus* solution experimentally in practice. We monitored the test *OpenStack* platform at KIS. We deployed the *Prometheus* server itself on one machine together with the *Grafana* visualization solution. To monitor the overall state of the CC platform, we deployed the *openstack exporter* plugin. We addressed the monitoring of the physical infrastructure using the *node exporter* plugin. In this deployment, we connected these plugins to the *Prometheus* server and visualized the collected data in the *Grafana* platform. To monitor the state of VMs in the CC platform, we used the *collectd* daemon, which, with the relevant plugins, collected statistics directly from the hypervisor on the compute node. We also connected the data from this daemon to the *Prometheus* server; however, due to lack of time, we were no longer able to visualize the data.

The result of this work is an experimental deployment, which we have thoroughly documented so that a real deployment on the new CC platform is simpler and faster. The documentation of the procedures in this work can further serve as an aid when deploying monitoring solutions, not only for CC platforms.

REFERENCES

[1] G. Aceto, A. Botta, W. De Donato, and A. Pescapè, “Cloud monitoring: A survey”, *Comput. Networks*, vol. 57, no. 9, pp. 2093–2115, 2013, doi: 10.1016/j.comnet.2013.04.001.

[2] S. Kundu, “Hybrid Cloud Workload Monitoring as a Service Hybrid Cloud Workload Monitoring as a Service A Project Report The Faculty of the Department of Computer Science San José State University Of the Requirements for the Class CS 298”, 2021.

[3] D. Tovarnňák and T. Pitner, “Towards multi-tenant and interoperable monitoring of virtual machines in cloud”, *Proc. - 14th Int. Symp. Symb. Numer. Algorithms Sci. Comput. SYNASC 2012*, pp. 436–442, 2012, doi: 10.1109/SYNASC.2012.55.

[4] J. Shao, H. Wei, Q. Wang, and H. Mei, “A runtime model based monitoring approach for cloud”, *Proc. - 2010 IEEE 3rd Int. Conf. Cloud Comput. CLOUD 2010*, no. July 2010, pp. 313–320, 2010, doi: 10.1109/CLOUD.2010.31.

[5] A. Kertesz *et al.*, “Enhancing Federated Cloud Management with an Integrated Service Monitoring Approach”, *J. Grid Comput.*, vol. 11, no. 4, pp. 699–720, 2013, doi: 10.1007/s10723-013-9269-0.

[6] G. Da Cunha Rodrigues *et al.*, “Monitoring of cloud computing environments: Concepts, solutions, trends, and future directions”, *Proc. ACM Symp. Appl. Comput.*, vol. 04-08-Apri, no. March 2018, pp. 378–383, 2016, doi: 10.1145/2851613.2851619.

[7] J. Montes, A. Sánchez, B. Memishi, M. S. Pérez, and G. Antoniu, “GMonE: A complete approach to cloud monitoring”, *Futur. Gener. Comput. Syst.*, vol. 29, no. 8, pp. 2026–2040, 2013, doi: 10.1016/j.future.2013.02.011.

[8] M. Drozdík, “Diplomová práca - Implementácia nástrojov na prácu so systémovými logmi a ich vizualizácia v prostredí katedry KIS”, p. 79, 2017.

[9] M. Dočár, “Diplomová práca - Zber a archivácia logov na KIS UNIZA”, p. 81, 2017.

[10] L. Chen, M. Xian, and J. Liu, “Monitoring System of OpenStack Cloud Platform Based on Prometheus”, *Proc. - 2020 Int. Conf. Comput. Vision, Image Deep Learn. CVIDL 2020*, no. Cvidl, pp. 206–209, 2020, doi: 10.1109/CVIDL51233.2020.0-100.

[11] “Prometheus documentation”. https://prometheus.io/docs/introduction/overview/.

[12] “Prometheus SNMP”. https://linuxhint.com/monitor-network-devices-prometheus/.

[13] “OpenStack exporter for Prometheus”. https://github.com/openstack-exporter/openstack-exporter#readme.

[14] “Collectd virt plugin documentation”. https://collectd.org/wiki/index.php/Plugin:virt (cited Feb. 01, 2023).

[15] “Collectd Write Prometheus plugin documentation”. https://collectd.org/wiki/index.php/Plugin:Write_Prometheus (cited Feb. 01, 2023).