PROJECT

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

Linked to work package KPB1 – Design of the virt. solution
Linked to output
V3 – Description of the selected solution meeting the defined criteria
type – documentation

CONTENTS

Introduction 7
2 Monitoring in a cloud computing environment 8
2.1 Cloud Computing 8
2.2 Motivation for monitoring 8
2.2.1 Capacity and resource planning 9
2.2.2 Capacity and resource management 9
2.2.3 Data centre management 10
2.2.4 SLA management 10
2.2.5 Billing 10
2.2.6 Troubleshooting 11
2.2.7 Performance management 11
2.2.8 Security 11
2.3 Perspectives on monitoring 12
3 Service level agreement issues in cloud computing 14
3.1 Components of the service level agreement 14
3.1.1 Basic components 14
3.1.2 Management components 15
3.1.3 Service attributes 15
3.1.4 SLA QoS metrics 15
3.1.5 Legal components 16
3.2 SLA metrics for IaaS Cloud Computing 16
3.3 Service level indicators 17
3.4 Service level objectives 17
3.5 Summary of SLA, SLI and SLO issues 18
4 Definition and issues of metrics 19
4.1 Metrics from the customer's perspective 19
4.2 Metrics from the provider's perspective 20
4.2.1 Network layer metrics 21
4.2.2 Hardware layer metrics 23
4.2.3 Operating system layer metrics 25
4.3 Measurement of selected metrics 25
4.3.1 Processor 26
4.3.2 Main memory 27
4.3.3 Storage 28
4.3.4 Network devices 29
5 Monitoring of the *OpenStack* platform 31
5.1 OpenStack 31
5.2 Monitoring in OpenStack 32
5.3 Issues of multi-tenant architecture in the context of monitoring 34
5.4 Presenting the collected data to stakeholders 35
5.4.1 Reports 35
5.4.2 Access to monitoring data – plotting metrics 37
5.4.3 Access to monitoring data – alerting 38
6 Survey of suitable tools for the solution 40
6.1 General overview of tools 40
6.1.1 Nagios 40

6.1.2 Graphite 40
6.1.3 Collectd 41
6.1.4 Grafana 41
6.2 Zabbix 41
6.3 Prometheus 42
6.4 Summary 43
7 Design of the technical solution 44
7.1 Requirements for the solution 44
7.2 Selection of technologies and hardware requirements 45
7.2.1 Prometheus server 46
7.2.2 Tools for metric collection 46
7.2.3 Visualization and alerting 48
7.3 Solution architecture 48
7.3.1 Central server 48
7.3.2 Other components 51
7.3.3 Overall architecture 52
7.4 Deployment 53
7.4.1 Prometheus server 54
7.4.2 Node exporter 55
7.4.3 Snmp exporter 55
7.4.4 Openstack exporter 57
7.4.5 Collectd 58
7.4.6 Grafana 59
7.4.7 Application as a service 60
7.4.8 *Prometheus* configuration and verification 61
7.5 Preparation for visualization 63
7.5.1 Connecting Prometheus and Grafana 64
7.5.2 Importing predefined templates 64
7.5.3 Folders 65
7.5.4 Creating users 65
7.5.5 Creating dashboards and panels 66
7.6 Monitoring dashboard for clients 68
7.7 Monitoring dashboard for the administrator 70
7.7.1 Visualization of the monitoring of *OpenStack* services 70
7.7.2 Visualization of reserved and available resources for the cloud 72
7.7.3 Visualization of the monitoring of the physical infrastructure 73
7.7.4 Other elements of the monitoring dashboard 75
7.8 Alerting 76
7.8.1 Alert rules 76
7.8.2 Contact points 77
7.8.3 Notification policies 78
7.9 Testing 78
7.9.1 Resource utilization and availability 79
7.9.2 OpenStack services 83
7.9.3 Suspicious VM behaviour 83
Conclusion 85
References 86

Annexes 91

LIST OF FIGURES

Figure 2.1 Perspectives on monitoring [5] 12
Figure 4.1 Network performance metrics [13] 22
Figure 4.2 Round-trip delay measurement [30] 29
Figure 5.1 OpenStack map [33] 31
Figure 5.2 Overview of responsibilities in CC services [46] 35
Figure 6.1 Prometheus architecture [60] 43
Figure 7.1 Architecture of the central server 49
Figure 7.2 Communication of openstack-exporter with the OpenStack API 50
Figure 7.3 Communication of snmp_exporter with network devices 51
Figure 7.4 Architecture of the other components 51
Figure 7.5 Topology of the monitored cloud 52
Figure 7.6 Monitoring server environment 52
Figure 7.7 Overall architecture 53
Figure 7.8 Snmp exporter interface 57
Figure 7.9 Beginning of the listing of metrics obtained using *snmp exporter* 57
Figure 7.10 Grafana web interface 60
Figure 7.11 *Prometheus* interface - jobs 63
Figure 7.12 Grafana adding a data source 64
Figure 7.13 *Grafana* creating a dashboard - navigation bar 66
Figure 7.14 Description of the dashboard interface 67
Figure 7.15 Example of part of the dashboard for clients – environment information 68
Figure 7.16 Example of part of the dashboard for clients – instance information 69
Figure 7.17 Details of the cinder service 71
Figure 7.18 Visualization of the monitoring of OpenStack services 72
Figure 7.19 Used resources in the cloud 73
Figure 7.20 Visualization of used resources on nodes 74
Figure 7.21 Overview of devices in the infrastructure 74
Figure 7.22 Verification of the *Sochor_Monitoring_Test* project 78
Figure 7.23 Adding a compute node 79
Figure 7.24 System load - *htop* 80
Figure 7.25 System load - Grafana 80
Figure 7.26 Created alerts - panel for active alerts 81
Figure 7.27 Generated alert - discord 81
Figure 7.28 Node unavailability - monitoring dashboard 82
Figure 7.29 Alert - node outage 82
Figure 7.30 Outage of the cinder service 83
Figure 7.31 Alert of the cinder service outage 83
Figure 7.32 Pending instance load alert 84
Figure 7.33 Active instance load alert 84

LIST OF TABLES

Table 3.1 SLA QoS metrics for IaaS [6], [9] 16
Table 4.1 *CloudWatch* instance metrics [11] 20
Table 5.1 Alert levels [52] 39
Table 7.1 List of components on the central server 50
Table 7.2 List and location of the deployed exporters 61
Table 7.3 List of rules 77

LIST OF ABBREVIATIONS

API – Application programming interface
CC – Cloud Computing
CLI – Command line (*Command line interface*)
CPU – Central processing unit (processor)
CS – Cloud service (*Cloud Service*)
CSP – Cloud service provider (*Cloud Service Provider*)
GB – Gigabyte
HTTP – Hypertext Transfer Protocol
IaaS – Infrastructure as a service (*Infrastructure as a Service*)
IT – Information technology
KPI – Key performance indicator (*Key performance indicator*)
MIB – Management information base (Management Information Base)
MS – Monitoring server
NPMs – Network performance metrics (*Network Performance Metrics*)
OS – Operating system
QoS – Quality of service (*Quality of Service*)
RAM – Main memory
RU – Controller node
SLA – Service level agreement (*Service Level Agreement*)
SLI – Service level indicator (*Service Level Indicator*)
SLO – Service level objective (*Service Level Objective*)
SNMP – Simple Network Management Protocol
SZ – Network devices
URL – Uniform resource locator (*Uniform Resource Locator*)
vCPU – Virtual central processing unit
VM – Virtual machine (*Virtual Machine*)
VU – Compute node

INTRODUCTION

The main objective of this document is the design and implementation of a monitoring system for a cloud built on the *OpenStack* platform, which provides infrastructure as a service. Within the requirements, it is essential that the system enables the measurement of the technical parameters of the cloud and the network infrastructure. This information should be presented in such a way that the provider is able to assess any threat to the provision of services. The system should also present information on the status and utilization of customer environments in a suitable manner.

To achieve our objective, it is essential to identify the motivation for monitoring, to distinguish the various perspectives on this process and to become familiar with SLA issues. We should also examine ways of extracting and presenting significant metrics from the infrastructure environment and the *OpenStack* platform. An important factor is also the examination of the available solutions that we can later use in the practical part. As part of the implementation itself, it is essential to design an architecture on the basis of which we will be able to deploy the system and test its functionality.

1 MONITORING IN THE CLOUD COMPUTING ENVIRONMENT

To begin with, it is appropriate to define what the term *Cloud computing* actually represents. In this chapter, we will attempt to identify the motivation behind monitoring such a system and we will distinguish between different perspectives.

1.1 Cloud Computing

For an explanation, we can draw inspiration from one of the current leaders in the provision of CC services, Amazon. In general, CC is the provision of information technology (IT) resources over the internet and on demand. It is characterized by billing based on the principle of paying for what you use (*pay-as-you-go*). Customers therefore do not need to manage their own physical infrastructure, but can access computing resources, storage and other necessary IT infrastructure that the provider is able to provide to them [1].

The main benefits of CC are said to be agility, elasticity and cost savings. Depending on the service provided, there are three main types of service models [1]. We can define them as:

- **Infrastructure as a service (IaaS)** – It is also commonly defined as the basic layer of CC. It typically provides access to network, compute and data (storage) resources, while giving the user the highest level of flexibility and control over IT resources among the types of CC. It is precisely this type that is the most important from the perspective of this work.
- **Platform as a service (PaaS) –** It removes the need for the customer to manage the infrastructure. In this case, the provider directly supplies a platform on which applications can be developed.
- **Software as a service (SaaS) –** For the ordinary internet user, this type of CC is the best known. It mainly concerns end-user applications (for example, a web email client) [1].

1.2 Motivation for monitoring

Both the provider and the user of such a platform have an interest in monitoring the CC environment. It is evident that the service operator needs to detect failures in both the physical and the virtual infrastructure. Its interest should also be to optimize the operation of the infrastructure [2].

Since the relationship between the provider and the customer should be formally defined, in this case in a service level agreement (SLA), the monitoring system should be able to alert the system owner to any problems that may affect the performance of the service and thereby cause financial damage. SLAs are discussed in more detail in Chapter 2.

The customer has an interest in monitoring the resources provided to them, mainly in order to have an overview of whether they need the allocated amount, since a larger amount of allocated resources means higher costs. On the other hand, they must know whether the resources allocated to them still meet the requirements of their project and whether they need to be increased [2].

There is a large number of processes that are critical for the operation of a CC platform, and for each of them monitoring is a key task. These processes, as well as the role of their monitoring, are presented below [3].

1.2.1 Capacity and resource planning

Before the mass adoption of CC, one of the greatest challenges for application operators and developers was the planning and management of the physical resources on which the given application ran. In order to guarantee a certain quality of service, they needed to quantify how many resources they required for the smooth running of processes. In general, such an estimate can be produced on the basis of testing, monitoring and statistical analysis, but the real values may be unpredictable and highly variable. In this respect, CC is a great advantage for application service providers, since this entire process of monitoring and managing the infrastructure is transferred to the operator of the CC system. Here we come to the fact that we also mention above in Chapter 1.2. Monitoring the underlying infrastructure is critical from the perspective of the CC service provider if it wants to track and predict the development of the parameters involved in the process of maintaining the quality of service (QoS). On the basis of the outputs from such a monitoring system, it should be able to plan correctly the development of the physical infrastructure and of the provided resources so that the SLAs concluded with all customers are respected [3].

1.2.2 Capacity and resource management

The basic prerequisite for a functional monitoring system is the ability to capture the current state of the CC platform. The main domain in CC is virtualization, which conceals the heterogeneity of the underlying physical infrastructure and thereby brings a higher level

of complexity for the provider, who consequently needs to manage both the physical and the virtual infrastructure. Virtual resources can migrate between physical devices, which may cause variability in the resources available on individual physical devices. A correctly deployed monitoring system is able to alert the administrator if there is a shortage of computing resources on any of the nodes. In addition, the aspect of service availability is important for many customers. Monitoring is able to reveal anomalies in the operation of the system, and we therefore consider it necessary in the context of ensuring the highest possible availability [3].

1.2.3 Data centre management

Cloud services are usually provided on the basis of large data centres, the management of which is an important task for the smooth operation of a complex system. Data centre management activities can be divided into two basic tasks:

- **Monitoring** – Tracking the required hardware and software metrics.
- **Data analysis** – Processing the collected metrics in order to evaluate the state of the system.

It is important that such a monitoring system enables a large number of operations in real time, since a large number of infrastructure elements is expected to be present in a data centre. In this context, the main motivation for monitoring is energy saving, i.e. optimizing the operation of physical devices [3].

1.2.4 SLA management

Monitoring is a mandatory and irreplaceable part of the process of verifying SLA compliance, for example when audit activities are carried out to determine regulatory compliance (e.g. where the service is offered to state authorities). Data from the monitoring system can help the provider formulate SLAs more realistically and design a sustainable billing model [3].

1.2.5 Billing

As already mentioned above, one of the basic characteristics of CC is the provision of services on a "pay for what you use" basis. This enables the customer to pay in proportion to their use of information resources. The basic prerequisite for billing in this way is the existence of a process that tracks the amount of resources used by customers. This process can be either separate or combined into one with the monitoring solution [3].

1.2.6 Problem detection

CC infrastructure is often complex and poses a challenge when it comes to precisely determining the origin of a problem that has arisen. When a problem occurs, it is necessary to analyse several possible components that may have failed. A further fact that adds to the complexity is that the components themselves may consist of several layers (e.g. the physical or virtual layer, the operating system (OS) of the host or of the virtual machine (VM), etc.). A comprehensible and detailed monitoring system can help the provider diagnose the problem that has arisen more quickly and more precisely. In addition, the customer should be able to verify whether a possible fault lies on their side or on the operator's side [3], [4].

1.2.7 Performance management

In practice, some compute nodes in CC deliver more performance than others. This has no fundamental impact on the ordinary customer, since the load is usually distributed evenly and the drop in performance is usually not significant. However, there may be customers who provide critical services using CC. For them, every change in performance is significant, and the availability of computing resources is critical. For such services, the customer usually uses several different providers, choosing the current one (on which the service will run at that moment) on the basis of the delivered, perceived performance. In this case, the customer has an interest in monitoring their perceived performance of the service, while the provider wants to provide them with the most reliable resources possible. In this case, monitoring is a necessity for both parties [3].

1.2.8 Security

Security in CC is important for several reasons. First of all, it is one of the main factors that deter customers from migrating their infrastructure to such an environment. To manage security in a CC infrastructure or in applications, a proper monitoring system is needed. When providing services to certain state organizations, CC must meet strict criteria concerning regulations on data management. This can be achieved by means of a monitoring system that is able to prove credibly that the user's data have not left the borders of the country [3].

1.3 Perspectives on monitoring

For the practical deployment of the solution, we need to determine the information that the system will collect. We primarily look at the issue from two perspectives. We describe them below.

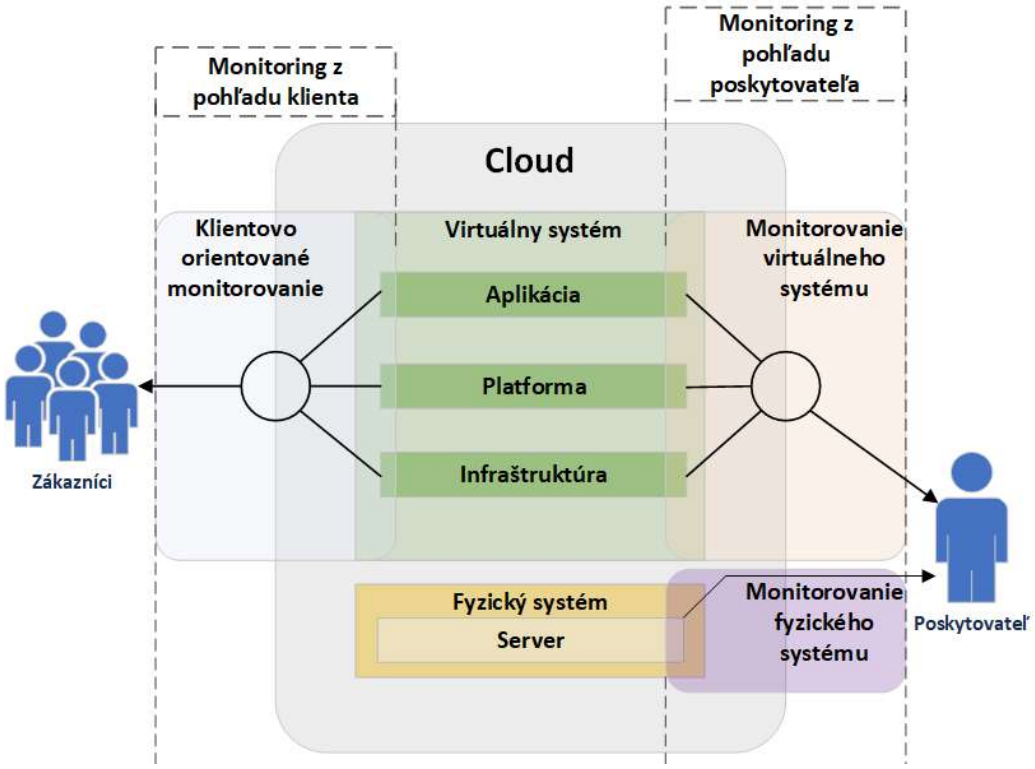


Figure 1.1 Perspectives on monitoring [5]

Figure 1.1 graphically shows what should, in general, be monitored and to whom the data from the individual parts should be presented. From the point of view of this work, neither the platform part nor the application part is of interest. We can see that the cloud service provider (CSP) should have information about the state of the physical system and the virtual infrastructure, which in the case of an IaaS platform means mainly VMs. The customer has no motivation to track the physical system and is therefore interested only in the state of their virtual resources. The provider may make such information available to them in some way.

Since the CC model is always closely linked to the guarantee of the quality of service, useful monitoring data should be linked to the conditions defined in the SLA (see Chapter 2). Defining metrics for both forms of monitoring, from the perspective of both the client and

the provider, is a critical aspect of a monitoring system. However, there are several problems that need to be taken into account in this context [5].

In the case of client-oriented monitoring, the metrics must provide information using the same parameters as defined in the SLA [5]. As an example, we give a scenario in which the CSP guarantees each customer a maximum storage capacity for their VM in gigabytes (GB). In such a case, the CSP must monitor the storage of each VM of all customers and then interpret this data appropriately in the same units.

The same principle applies to the monitoring of a virtual system. What should be monitored is also closely linked to the SLA. However, for the CSP, this data should provide additional information [5]. For a better understanding, we will build on the example in the previous paragraph. In addition to monitoring the storage on each VM, the CSP should also collect data on the total virtual storage in the CC platform. It should know the total storage utilization by all clients. This information should serve only the CSP.

The case of monitoring the underlying physical system differs from the previous two, since it concerns the lower level of the infrastructure on which the entire CC platform rests. This means that the required metrics are the same as for ordinary distributed systems. On the hardware side, these metrics include, for example:

- CPU utilization,
- RAM utilization,
- physical storage utilization and
- network traffic [5].

On the OS metrics side, we can include here, for example:

- system load and
- virtual memory utilization, since its management is the responsibility of the OS [5].

Information collected in this way serves primarily for the management of the entire system. In this case, the link with the SLA is rooted in the relationship between the low-level physical system and the high-level virtual system that is built on the given infrastructure [5]. We will discuss metrics in this work in more detail in Chapter 3.

2 SERVICE LEVEL AGREEMENT ISSUES IN CLOUD COMPUTING

Since in the previous chapter we identified SLA management as one of the motivations for creating a monitoring system, in this chapter we will address this issue in more detail. We can define an SLA as a format containing a description of the agreed services, service level parameters and guarantees regarding the quality of service (QoS). An important part of this document is also the part dealing with penalties in the event of a breach of any of the agreed parts. Penalties are mostly stated in a quantifiable manner. An SLA forms a significant contract between the service provider and the customer, who interacts in some way with the service provided. A key aspect of an SLA is to provide a clear definition of the basic parts of the service, such as performance, availability, billing, etc. This format is used in a wide range of sectors. In the IT industry, these include, for example, the provision of web services, data centre management, networking sectors, etc. The specific wording and description of an SLA may vary between individual sectors, while the basic idea remains the same [6].

2.1 Components of the service level agreement

An SLA contains components that, for simplicity, we can classify into individual categories. Thanks to them, the processing of a new SLA can be more systematic, which facilitates the actual process of creating such a document. In simplified terms, we can sort these components into five categories, which we present below together with the most important ones from the perspective of SLAs in CC [7].

2.1.1 Basic components

The basic components of an SLA include:

- **Subject of the SLA** – Defines why the given SLA is being drawn up and which parties are involved in it.
- **Effective date and expiry date** – The time interval during which the given document is valid. If both parties are interested, this interval can be extended.
- **Version number** – If the parties concerned agree on any change in the wording of the SLA, the version number must be adjusted in the agreed manner, for example by incrementing the value [6], [7].

2.1.2 Management components

The management components of an SLA include:

- **Reporting format and frequency** – This part defines in what way, in what format, in how much detail and how often reports will be provided, whether by the user or by the customer themselves.
- **Payments and compensation** – The customer must make payments in accordance with the conditions laid down in this part. Compensation from the provider in the event of a breach of the SLA is also determined here [7].

2.1.3 Service attributes

The service attributes include:

- **Service identification** – In this part, the service is uniquely defined [6], [7].
- **Scope** – This part determines the domain or scope in which the service exists. Service level objectives (SLOs) and key performance indicators (KPIs) of the service are also mentioned here [7].
- **Environment** – This part describes the environment in which the service operates most effectively. It may define how changes in the environment, or in configuration parameters, affect the effectiveness of the service [7]. In the case of IaaS CC, we can imagine the quality of the customer's internet connection as one of the environment attributes. If the customer had high latency or a low connection transfer rate, this could affect their perceived quality of service, even though no problems occurred on the provider's side. It is therefore advisable to include such attributes in the contract.
- **Exceptions to the quality of service** – In the context of operating the service, a situation may arise in which the provider is unable to provide the given service. This may include, e.g., regular maintenance, power supply outages, etc. Situations known in advance may be stated in this part of the agreement, and in the event of such outages the provider is exempt from the defined sanctions [7], [8].

2.1.4 SLA QoS metrics

- **Definition of metrics** – In this work, we will focus mainly on the areas defined in this part of the SLA. QoS indicators are described here by means of metrics. For the given service, various characteristics are analysed using mathematical methods, which are also referred to as service metrics (*Service Metrics*) [7]. From the perspective of this work, it is appropriate to define those that are relevant to the IaaS cloud service. We address this issue in more detail in Chapter 2.2.

- **Measurement methods** – The measurement methods used and the calculation mechanisms are usually stated in this part of the SLA [7].

2.1.5 Legal components

- **Authority** – States which authority can be approached in the event of any disputes.
- **Settlement procedure –** Stated mainly for the case that the service provider goes bankrupt.
- **Contract termination policy**
- **Confidentiality –** Legal definitions relating to the protection of sensitive data.
- **Compensation for damage**

2.2 SLA metrics for IaaS Cloud Computing

From the perspective of this work, we are interested in monitoring the IaaS cloud service. In order to determine the parameters to be monitored, we first need to define useful metrics for this cloud service from the SLA perspective. We can draw inspiration, for example, from Amazon, which provides, among other things, IaaS cloud services. Several sources [6], [9] agree that the most important metrics for this cloud service include those

Parameter	Description
Processor capacity (CPU)	Speed (and number of cores) of the processor for the VM
Main memory size	RAM main memory available to the VM
Start-up duration	Time until the VM is ready for use
Storage	Size of storage for short-term or long-term use
Scalable up	Maximum number of VMs per user
Scalable down	Minimum number of VMs per user
Scale-up time	Time to increase a certain number of VMs
Scale-down time	Time to decrease a certain number of VMs
Automatic scaling	Boolean value for support of the auto-scaling feature
Maximum number of VMs	Maximum number of VMs that the customer can launch
Availability	Usually a percentage expressing the proportion of time during which the service is available
Response time	Duration of receiving and executing a request

Table 2.1 SLA QoS metrics for IaaS [6], [9]

listed below.

2.3 Service level indicators

Such a document also includes service level indicators (SLIs). We will therefore also briefly address this issue below.

We define an SLI as a quantitative measurement of a certain aspect of the service level. As the name itself implies, an SLI indicates this aspect. We regard the resulting value of such a measurement as the SLI [10].

Measurements are often aggregated, i.e. raw data are collected during a measurement window and then converted into a ratio, average or percentile. Ideally, an SLI measures the service level directly. In some cases, however, this is either not possible, or obtaining the data is difficult and not cost-effective. Measuring latency on the client side falls into this category. From the SLA perspective, such latency is a much more significant metric, but in practice the provider can usually measure latency only on the server side [10].

A very important SLI is also availability, in other words the ratio of the time during which the service is available to the total time of service use. For every service, it is important to also allow for outages, so 100% availability is not a realistic assumption. In the IT industry, availability at the level of 99% and higher is often guaranteed [10].

2.4 Service level objectives

Like the SLI, in the context of an SLA it is also appropriate to define the term service level objective (SLO). An SLO represents the required value for the service level, which is measured by means of an SLI. How to choose an SLO and what value to set it to is a complex process [10].

Determining and publishing SLOs for users enables them to set their expectations of how well the given service performs. Thanks to such a strategy, the provider is able to reduce unfounded complaints, for example about slow service speed. If SLOs are not explicitly defined, users tend to form their own expectations regarding service performance, which may be unachievable for the provider. On the user's side, two opposing phenomena may arise on the basis of the facts mentioned above. On the one hand, there may be excessive reliance on the given service, which may have a negative consequence in the event of an outage. On the other hand, the user may come to believe that the service is not very reliable. From real events, the phenomenon of excessive reliance is the worse one from the provider's perspective, which is why an interesting solution is used in practice. After the SLO, for example for availability, has been set to a certain value (for the sake of

example, let us assume a value of 95% of the time), the service is deliberately shut down even in the case of 100% functionality, so that the prescribed value is met [10].

2.5 Summary of SLA, SLI and SLO issues

It is difficult to operate a service without understanding which attributes most affect its operation and how to measure and evaluate them. On the basis of intuition, experience and knowledge of what users expect from the service, the provider should be able to construct an SLA with defined SLOs that are verified by means of SLIs.

In other words, the provider should set the values it guarantees for the various aspects of the service (SLOs) and create measurements of these aspects, the results of which (SLIs) will be compared with the SLOs. On this basis, it is determined whether the SLA is fulfilled [10].

In the practical part, we expect the system to perform measurements of some of the metrics mentioned in Table 2.1, above all, however, of availability. From these measurements, the system will obtain the SLI value, which it will present and also compare with the preset SLO value. If SLI → SLO, the system will report a possible threat to the provision of the service in accordance with the SLA. If SLI < SLO, the system will present a breach of the SLA.

3 DEFINITION AND ISSUES OF METRICS

In the previous chapter, we attempted to define important metrics from the SLA perspective. Besides this, however, the customer may also be interested in the performance and utilization of their resources, so we need to identify metrics that are able to present such information. In addition, we are interested in identifying metrics presenting the health, performance and state of the entire system for the purpose of managing the operation of the platform. We therefore first need to know what the system consists of and then identify suitable metrics for its individual parts.

3.1 Metrics from the customer's perspective

In addition to the parameters guaranteed in the SLA, the customer is motivated to track the performance of the resources allocated to them. We, as the service provider, want to provide them with such information, which, however, we first need to identify. In our case, we will draw inspiration from an industrial solution by Amazon, see [11]. Their cloud platform (AWS) is used worldwide and provides customers with a wide range of cloud services (CS).

From the perspective of this work, we are interested in Amazon EC2 (*Amazon Elastic Compute Cloud*), which is an IaaS service from Amazon. It provides a wide range of virtual instances with the availability of various types of CPU, storage, operating system and the like [12]. It enables the customer to monitor their instances by means of a solution called *CloudWatch*.

The industrial solution mentioned provides several categories of metrics that are available to the customer. Most of these categories are closely related to Amazon's accounting system. These categories are not significant from the perspective of the solution in this work. However, we can draw inspiration from the VM instance metrics category, see Table 3.1, which contains mainly technical parameters. If of interest, the other categories are available online in the Amazon *CloudWatch* documentation, see [11].

We can state that the table below gives us a view of which metrics the proposed system could collect and present to the customer in addition to those we defined in connection with the SLA in Chapter 2.

Instance metrics		
Metric	**Description**	**Unit**
CPU utilization	Percentage of the allocated EC2 compute units that the instance uses.	Percent
Disk read operations	Completed read operations from all disks available to the instance.	Count
Disk write operations	Completed write operations to all disks available to the instance	Count
Disk read volume	Number of bytes read from all disks available to the instance.	Bytes
Disk write volume	Number of bytes written to all disks available to the instance.	Bytes
Volume of incoming traffic	Number of bytes received on each of the instance's interfaces.	Bytes
Volume of outgoing traffic	Number of bytes sent from each of the instance's interfaces.	Bytes
Number of packets received	Number of packets received on each of the instance's interfaces.	Count
Number of packets sent	Number of packets sent from each of the instance's interfaces.	Count

Table 3.1 *CloudWatch* instance metrics [11]

3.2 Metrics from the provider's perspective

It is relatively difficult to identify from the literature specific standards, procedures and metrics for monitoring CC as a whole. To begin with, we will therefore set out the layers by which we can model a cloud. Since the provider is interested in knowing all aspects of the platform it operates, these layers can give it a view of where to place monitoring stations. For completeness, we list below all seven layers as defined by the non-profit organization *Cloud Security Aliance* [3].

- **Facilities –** This layer includes the data centres in which the computing and network resources are housed [3]. In this work, we will not deal in more detail with the monitoring of this layer, but in the case of commercial provision of CS outside the

academic environment, it would be advisable to extend the monitoring system to include this part as well.

- **Network –** In this layer, we can find network devices and interconnections, both within the underlying physical infrastructure and (where possible) between the cloud and the user.
- **Hardware –** This layer is characterized by the physical components of both computing and network devices.
- **Operating system –** We consider as part of this layer the software components that make up the operating system on the physical hosts or on the virtual instances themselves.
- **Intermediary software (*middleware*) –** It is the software layer between the operating system and the user application, typical of SaaS and PaaS.
- **Application –** Likewise typical of SaaS and PaaS.
- **User –** We include in this layer the end user of the CS and applications running outside the CC (for example, a web browser running on the user's device) [3].

From the perspective of a provider of primarily IaaS services, we consider the facilities, network, hardware and operating system layers to be significant. We will therefore attempt below to identify significant metrics for these layers. We can also argue that the end user is an important layer as well, but the practical implementation of this type of monitoring is hardly feasible.

3.2.1 Network layer metrics

The network performance metrics (NPM - *Network Performance Metrics*) shown below correspond to the basic performance metrics in the network management layer. We can divide them into four types as follows:

- **Availability –** This includes the functionality of the physical connectivity of individual network elements and network devices.
- **Packet loss –** Put simply, it is the ratio of lost to sent packets between the sender and the receiver over a certain time interval, mostly expressed as a percentage. A one-way delay is defined, which expresses this ratio only in the direction from the sender to the receiver, and a round-trip delay, which also includes the path of the response in the opposite direction.

- **Delay –** Expresses how long a time interval it takes a packet to make a one-way or two-way trip. It mostly consists of three factors: one-way delay, round-trip delay and delay variation.
- **Utilization –** Is the percentage expression of the current load of a link in relation to the maximum throughput [13].

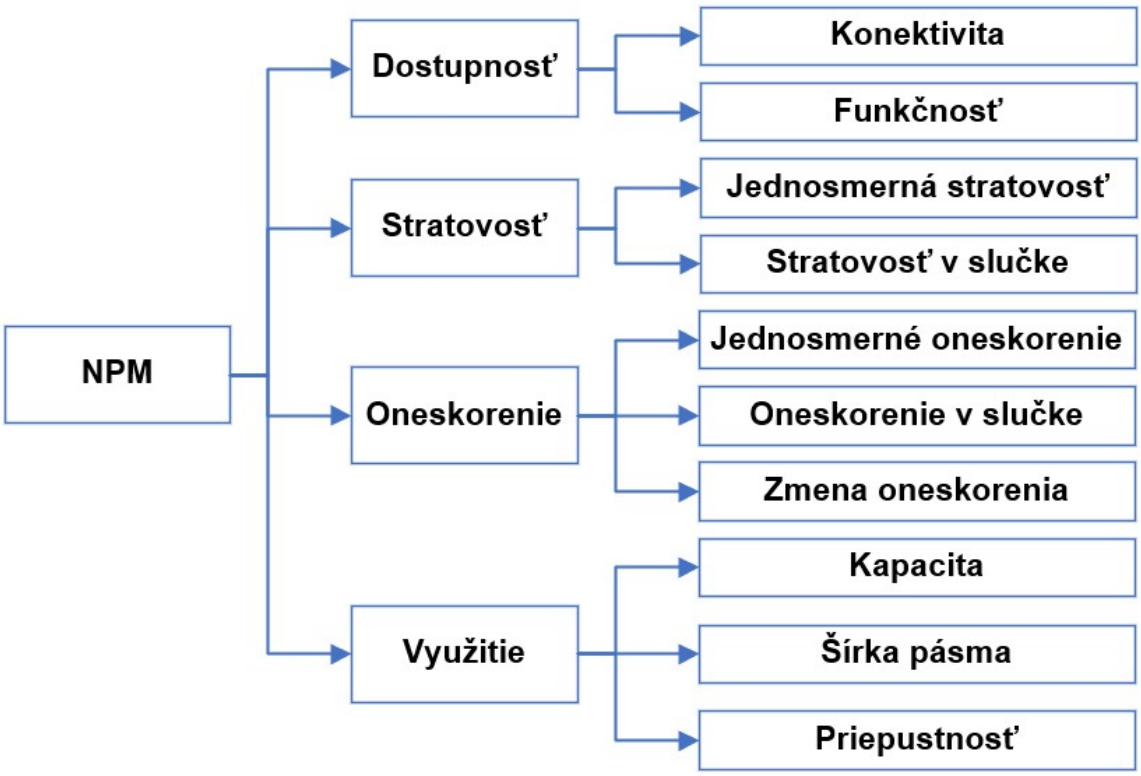


Figure 3.1 Network performance metrics [13]

Data centres are typical of the cloud, so we are mainly interested in the issue of monitoring their infrastructure, where the ideology of oversubscription (Eng. *oversubscription*) is common practice. Put simply, this means that the provider gives customers, in total, a larger amount of resources than their real maximum capacity. In practice, it is applied mainly for the purpose of maximizing available resources and increasing profitability. It is based on the low probability of an event in which all customers would use all their resources at one hundred percent. It follows from the above that the key objective is the ability to identify congestion and performance degradation quickly. For this purpose, the metrics of delay and CPU utilization of network devices in the affected network serve us most significantly [14].

3.2.2 Hardware layer metrics

This layer includes the physical devices and their components that make up the computing or network layer necessary for the operation of the CC platform. Below, we will attempt to identify which metrics are significant from the perspective of service provision and the operation of the CC platform.

Since the concept of CC has its roots in *grid* networks, or in distributed systems [2], metrics that are important from the perspective of such systems are also essential in CC. The Google SRE book [10] defines the four golden signals of monitoring, which express the most important factors that need to be tracked in any user-facing system. These characteristics are listed below:

- **Latency –** The time required to complete an action. Tracking the latency of components makes it possible to determine weak points or to identify an incipient failure.
- **Traffic –** By traffic we mean how busy a system or a system component is. This characteristic is mainly concerned with capturing the load on, or demand for, the system. Put simply, it expresses how much work the system is currently performing. By tracking component traffic, we can determine a shortage as well as a surplus of working capacity for a given component.
- **Errors –** Tracking errors is important in the context of knowing the overall health of the system. Based on error data, we can anticipate weaknesses of the system and determine how often individual components fail. In the event of a failure of the entire system, some errors can indicate where and why the system failed.
- **Saturation –** Measures the current utilization of a given resource. Ratios or percentages are often used in this context. Saturation data provide information on the load of the resources that a service or application needs in order to function efficiently [15]. A heavy load on components may lead to a drop in service performance.

We can use the signals above as a guideline, while we need to establish how these metrics will be expressed in the hardware layer of the system. The current technological trend is the abstraction of physical components and the low-level OS. It is an undeniable fact, however, that every service relies on the underlying hardware and OS for its functionality. If we want to identify which metrics are appropriate to collect and evaluate, we first need to ask a fundamental question: which individual components form the base layer for running the platform?

We know that the important parts are the hardware components of servers (CPU, memory, storage, network devices) and some basic abstractions provided by the OS (processes, file descriptors). If we look at each of these components in the context of the four golden signals, we can identify suitable metrics for monitoring.

In the context of the **CPU**, the following measurements are appropriate:

- **Latency –** Average or maximum latency within the CPU scheduler.
- **Traffic –** Overall CPU utilization.
- **Errors –** Processor-specific errors, CPU failure.
- **Saturation –** Run queue length.

In the context of **memory**, the individual signals may look as follows:

- **Latency –** For memory, it is difficult to find a suitable measurement method.
- **Traffic –** The amount of memory currently in use.
- **Errors –** Errors of the type *Ouf of memory.*
- **Saturation –** SWAP usage.

For **storage** we should consider the following measurements:

- **Latency –** Average wait time for writes and reads.
- **Traffic –** I/O levels for both reading and writing.
- **Errors –** File system errors, disk errors in */sys/devices*.
- **Saturation –** I/O queue size.

For **network devices**:

- **Latency –** Network driver queue length.
- **Traffic –** Incoming and outgoing bytes or packets per second.
- **Errors –** Network device errors, dropped packets.
- **Saturation –** Dropped packets, retransmitted segments.

Together with the metrics representing the state of physical resources, it is also appropriate to collect OS-related metrics [15].

3.2.3 Operating system layer metrics

In the case of a CC platform built on the *OpenStack* solution and providing an IaaS service, we consider it appropriate to include in this layer the OS running on the physical nodes as well as the cloud OS *OpenStack* itself.

As regards suitable metrics for an OS running on physical devices, it is difficult to identify them precisely from the literature. In general, this depends mainly on the requirements of the administrators. As a starting point, we can draw inspiration from the ***Utilization, Saturation*** *and* ***Errors*** method (USE – *Utilization Saturation and Errors*), which can be used to analyse the performance of any system [16]. We also drew on this method in the previous chapter for the hardware components. Within this method, the following are given as significant software metrics:

- **Process capacity** – The system has a limited number of processes and threads. In this context, it is appropriate to track their current utilization and the number of processes waiting to be processed, and possibly also to monitor allocation errors.
- **File descriptor capacity** – For which it is likewise appropriate to track their current utilization and any errors [16].

In addition to the metrics above, from the software perspective we also consider **server uptime** to be important, as it can indicate whether the system has shut down unexpectedly without the administrator's knowledge. We can define it as the amount of time during which the server is able to process requests. We calculate this metric as *Total time – downtime* [17].

The cloud OS *OpenStack* and those of its components that would be appropriate to monitor are discussed in more detail in chapter 4.2.

3.3 Performing measurements of selected metrics

It is beyond the scope of this work to address how to measure all the metrics presented in the solution so far. Therefore, below we define what we consider to be the most important metrics, together with possible ways of measuring and evaluating them. From the context so far, it is clear that the most important components are the CPU, memory, storage and network elements. It should be clarified here that we are motivated to collect information about these components from both the physical and the virtual layer, while the method of extraction may differ between the layers. In a physical system, the method of obtaining the data needed to calculate the metrics may vary depending on the type of OS. In this work, we discuss this topic in the context of Linux/UNIX systems.

From the perspective of the monitoring system's functionality, we consider it important that we do not have to access the virtual systems directly; therefore, we need to obtain the above-mentioned data in another way. We have concluded that we will obtain this information by communicating with the hypervisor. The *virsh* program is suitable for this purpose; for communication it uses the *Libvirt* toolkit (a virtualization API) written in C. More information about the program and the toolkit it uses is available on the official *Libvirt* manual pages, see [18].

For each component, we will try to identify which results may herald a problem, since later we will want to inform administrators about such events. We address this issue only in the context of physical systems.

3.3.1 Central processing unit

For the CPU, we consider its utilization to be the most important metric, i.e. the amount of time during which it was performing work, usually expressed as a percentage.

In a physical system, statistics on processor usage are stored by the kernel (*kernel*) in the designated file `/proc/stat`. An explanation of the format and meaning of the individual values is available in the Linux manual, section *proc(5)*, or online at [19]. What is important for us is that from these data we can identify how much time the CPU cores devoted to:

- The user environment.
- The system kernel.
- Waiting for I/O operations to complete.
- Idling (the processor was not performing any work).
- Servicing interrupts, and
- other activities (see [19]).

Based on the information provided in this way, we can calculate the proportion of time for which each core was in use. To calculate the utilization of the CPU as a whole, the arithmetic mean of the utilization of all its cores is sufficient [20].

In a virtual system, several methods are available for extracting information on virtual CPU (vCPU) utilization using the *virsh* program. For more information on the specific options, we refer the reader to the documentation of this solution (see [21]); we recommend looking up information on the `domstats` command and its arguments `cpu-total` and `vcpus`. Alternatively, the `cpu-stats` command can also be used. With any of these

options, however, we cannot obtain the idle time. We can, however, use the vCPU working time.

How high a utilization can cause a problem? In this respect, the most critical part is the CPU scheduler, or more precisely low-priority tasks, which under high utilization may never get access to computing resources. In general, the upper limit for long-term utilization is given as between seventy and eighty per cent. Processors that are continuously loaded above this threshold may exhibit unpredictable behaviour. System designers are usually advised to aim for a maximum processor utilization below fifty per cent when designing new systems [22].

3.3.2 Main memory

As with the CPU, for RAM in a physical system we consider its utilization to be the most important metric. The process of extracting the basic data is also very similar. Memory usage statistics are stored in the system file `/proc/meminfo`. More information is again available in the Linux manual; below we list only a few basic statistics that the system offers us in this way.

- **Total memory** – Total usable RAM.
- **Free memory** – The amount of free memory in the system. This does not include caches that can be cleared.
- **Available memory** – Free memory; this value also includes caches.
- Usage of various types of **caches**.

From a practical point of view, of all the statistics offered, the first two mentioned are the most important. We calculate the amount of used memory as the difference between total and free memory [19], [23].

For a virtual system, too, we will proceed similarly to the CPU. *Virsh* offers us the `dommemstat` command, which returns memory statistics for a selected running VM. Thanks to this, we can obtain, among other things, data on total, used and free memory. All the information that can be obtained with this command, its units and description are likewise available in the official documentation [21].

Determining how high a main memory utilization can cause a problem is more difficult to identify than it was for the CPU. The recommendations for GitHub Enterprise Server specify the first form of lower-severity alert (more on alert severity in 4.4.3) for a long-term load above fifty per cent. For a sustained load above seventy per cent, they recommend a higher-severity alert [24]. Other sources specify reporting a problem at utilization above

seventy-five or even ninety per cent [25], [26]. Within memory utilization alerts, it is also appropriate to have an alert set for the use of swapping, i.e. moving data from main memory to disk in order to free up capacity. Some sources suggest that alerting is appropriate only at seventy per cent swap usage, but from our practical experience we consider this threshold to be too high and recommend setting it at around twenty per cent, since the swapping process is demanding on system resources and wears out SSDs, which have a limited number of write cycles.

3.3.3 Storage

In Linux systems, the `df` command is used to display information on disk usage by individual file systems. As in the previous cases, its use is described in the Linux manual (see [27]). Interestingly, storage size is not the only limiting factor in this case. There is also the inode, which represents a data structure holding metadata about directories and files. Put simply, it represents the number of files stored in a given file system. Their number is also limited, but in practice the maximum of this structure is rarely exceeded before the storage capacity limit is reached. The `df` command allows us to extract the *Available* value itself, which represents the remaining free space in the given file system [27]–[29].

Virsh also allows us to obtain statistics on the disks assigned to a given VM. Again, the `domstats` command with the argument `block` is used for this purpose. For a more detailed description, see [21].

Filling up storage is a slower process, but it should certainly not be given less importance. Setting an alert is appropriate when less than 10% of free space remains on the disk. If the monitoring system supports it, it is useful for it to be able to estimate, based on previous trends, when the device will run out of free capacity and to report this figure to the administrator [26].

3.3.4 Network elements

In chapter 3.2.1, we identified the most important metrics for monitoring the network layer, namely latency, link load and CPU utilization on devices within the data centre network. We will first address latency measurement and then move on to monitoring the CPU and other components on network devices. ***Latency*** is divided into one-way and round-trip latency. We consider the latter to be the metric providing greater informational value, and we will therefore focus on this type. There are two ways of measuring it, active and passive. Simply put, with the active method we generate new traffic, which we then monitor. With the passive method we generate nothing extra but obtain information from already existing traffic. The active method is simpler to implement, but care must be taken that the volume of generated traffic does not have a significant impact on the existing traffic. We define round-trip latency as the difference between the time the first bit of the request is sent and the time the last bit of the response is received on the same host [30]. The measurement procedure is shown graphically in **Figure 3.2**.

In the world of networking technologies, the Simple Network Management Protocol

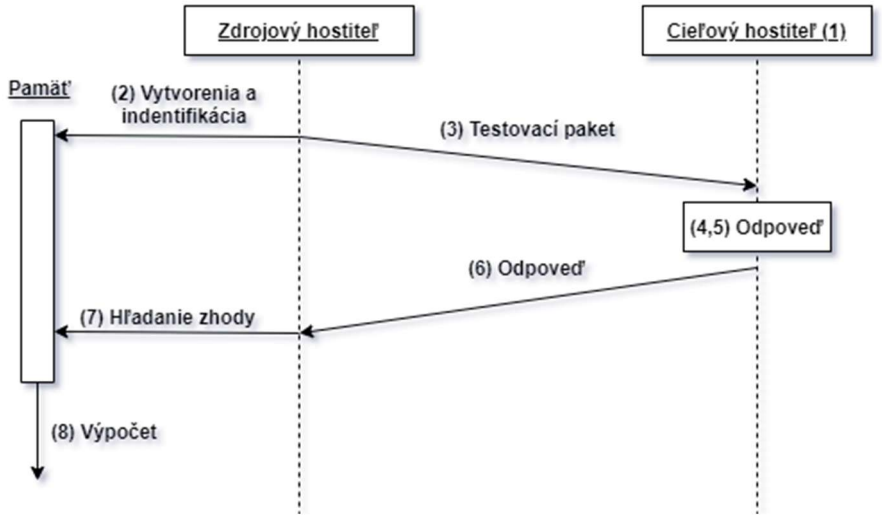


Figure 3.2 Round-trip latency measurement [30]

(SNMP) is widely used and well established for managing and monitoring devices. This protocol classifies the supported metrics into a hierarchical data structure called the Management Information Base (MIB). The module present on the managed element, otherwise known as the SNMP agent, responds to queries from the managing element, the SNMP manager. The response to such a query consists of the values of the requested object in the MIB, corresponding to its type. Thanks to SNMP, we can extract from network devices the CPU utilization mentioned above, as well as other passive metrics, such as the link load or queue length mentioned earlier [31].

Since network elements are physical devices, we do not need to address the virtual environment in this section. We will mention, however, that we could use SNMP or direct outputs from the software components that emulate such infrastructure in a virtual environment. In the context of *OpenStack*, this is the software called *Neutron*. For more information on monitoring virtual network infrastructure, we refer the reader to [32].

As regards alerting, the same philosophy applies to the CPU as discussed in chapter 3.3.1. For latency, it is problematic to define an exact threshold from which we speak of a problem. Simply put, the higher the latency, the worse the system performance. A suitable solution could be to track the latency value over a certain time window, use the data from such a measurement to evaluate the average latency in the system, and alert to a possible problem when two or three times this value is exceeded.

4 MONITORING OF THE *OPENSTACK* PLATFORM

For a proper understanding of the topic, it is necessary first to become briefly acquainted with it.

4.1 *OpenStack*

OpenStack is a cloud operating system capable of controlling large amounts of compute, storage and network resources. To manage these resources, it uses an application programming interface (API) with common authentication mechanisms. In addition to the basic type of service, which is infrastructure, this OS offers, by means of add-on components, capabilities for orchestration, failure prevention and service management to ensure high availability of user applications [33].

Originally, it included only two basic modules, namely:

- **Nova** – Developed at NASA, it provides a way to ensure the creation, running and termination of VMs [34], [35].
- **Swift** – Developed by *Rackspace, it* manages data storage [34], [36].

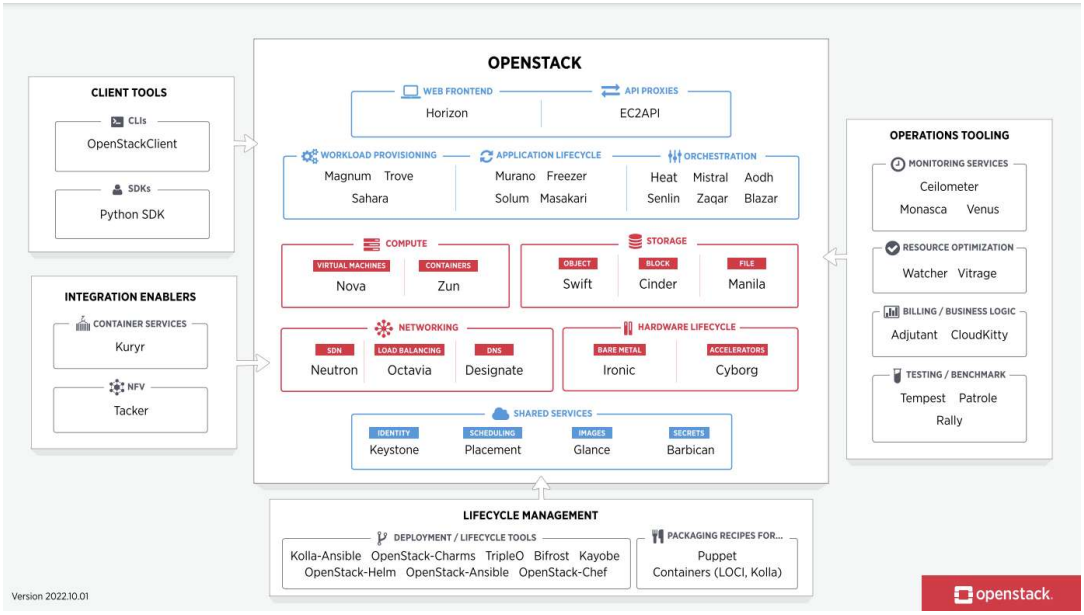


Figure 4.1 OpenStack map [33]

With the involvement of manufacturers of servers, networking equipment, storage and other giants in the IT world, the *OpenStack* community is very active, and the whole system is developing at a rapid pace. In general, it separates users from massive hardware resources behind the network, via the network. It is responsible for interaction with the

hypervisor on the physical system [34]. This system currently comprises a large number of components, each of which fulfils its role in the context of the whole system. The names of the most important components, together with their roles, can be seen on the *OpenStack* map shown above.

4.2 Monitoring within OpenStack

From our point of view, the data collected directly from *OpenStack* are very important, as they are closely linked to SLA metrics. We have already discussed SLAs in chapter 2; here we will try to link this topic with *OpenStack*.

We already know that with an IaaS service the CSP provides the customer with computing resources, which can be divided into three main categories:

- **Compute resources –** This category primarily includes the CPU and RAM. In *OpenStack*, the management of this category is the task of the ***Nova*** service. It is precisely this service that gives us a view of SLA parameters such as processor capacity, main memory size and the maximum number of VMs per customer.
- **Storage –** In *OpenStack*, storage management is entrusted to the ***Cinder*** service. From it we can identify another SLA parameter, the size of the allocated storage.
- **Network –** Network elements are managed by the ***Neutron*** service. Although we cannot directly identify any of the parameters specified for the SLA (see Table 2.1) that we could extract from this service, its functionality is critical, since in the event of its outage no communication is ensured for instances in the cloud, which may appear to the customer as an outage in the availability of their resources.

In addition to the categories mentioned above, we identify two further important components, namely ***Glance*** for managing the available operating systems. ***Keystone*** is used for authentication and authorization within the platform [37].

So what is it appropriate to monitor from *OpenStack*? From our point of view, we see the need to monitor the availability of all five components listed above. The reason is that an outage of any of them disrupts the functioning of the service. A potential SLA could even specify the availability of the individual components separately. Moreover, each component consists of parts that have their own specific tasks. We propose monitoring these parts as well, if possible. An overview of the most important ones is given below; for some of them we briefly describe the impact their outage may have on the platform.

- **Compute service (nova)**

- ***nova-api*** – Accepts HTTP requests, which it then converts and passes on to the other parts. In the event of an outage, no component will be able to communicate with the *nova* service.
- ***nova-scheduler*** – Decides which host is allocated which instance. In the event of an outage, the creation of new instances may be affected.
- ***nova-conductor*** – Handles requests that require coordination between the other parts or the database. An outage may render the component as such non-functional.
- ***nova-compute*** – Manages communication with the hypervisor and VMs. In the event of an outage, instances may become non-functional [38].

- **Storage service (cinder)**
  - ***cinder-volume*** – Manages dynamically attachable block devices. In the event of its outage, the ability to create or modify block storage is lost.
  - ***cidner-api*** – Has the same role as nova-api, but for the storage component.
  - ***cinder-scheduler*** – Decides which host is allocated the individual volumes [39].
- **Networking service (neutron)**
  - ***neutron-server*** – Accepts API requests and routes them to the appropriate *OpenStack* networking plug-ins for action. It follows that when it is non-functional, the functionality of the cloud's network layer is lost.
  - ***OpenStack networking plug-ins and agents*** – They are responsible for plugging and unplugging ports, creating networks and subnets, providing IP addressing, etc. Their number and type depend on the provider and the technologies deployed in the given environment. Common ones include, for example, the L3 agent and the DHCP (dynamic IP address allocation) agent [40].

OpenStack itself offers its own monitoring project called *Telemetry*. Its aim is to reliably collect data on the utilization of physical and virtual resources, as well as of the above-mentioned components within the deployed cloud. It also offers capabilities for analysis and for triggering actions when defined criteria are met. As stated in the official documentation, this project consists of further smaller projects, which are:

- **Ceilometer** – responsible for collecting, normalizing and transforming the data produced by *OpenStack* components, and
- **Aodh** – which makes it possible to trigger an action based on defined rules [41].

In addition to those mentioned above, there is also a solution called *Monasca*, which provides monitoring as a service and can also integrate with *OpenStack* [42].

4.3 The issue of multi-tenant architecture in the context of monitoring

Multi-tenant architecture is the natural result of efforts to achieve a positive economic effect based on economies of scale. In the context of CC, this effect is achieved through the use of virtualization and the subsequent sharing of resources [43]. A client, in this context also called a *tenant,* can be any application, user or group of users that requires its own secure and exclusive virtual environment [44]. In the case of monitoring a CC platform, a major challenge is to identify what level of access to monitoring data to guarantee to individual clients. Two factors that arise with this architecture in the context of access to the collected data need to be considered. These factors are:

- **Concurrency –** In some cases (especially with PaaS), several clients are interested in accessing the same data. In such a situation, the provider must appropriately provide several clients with access to identical information from the monitoring system [45]. From the IaaS perspective, it is difficult to identify precisely from the literature when such a situation could arise. Theoretically, we could assume that several customers would be interested in data from the same physical node on which their virtual environment runs.
- **Isolation -** This factor is very important for the monitoring system, as it forms a large part of the customer's trust in the provider. Clients must not have access to information collected from environments intended exclusively for other clients [45]. Within this work, this means that a client must have access to information only about those VMs that actually belong to them.

For completeness and to link the terms CSP and *tenant*, see **Figure 4.2**. It provides an overview of which party should assume responsibility for the individual parts in the basic types of CC services. These parts include:

- **Infrastructure –** The hardware and OS belonging to the devices on which the platform runs.
- **OS –** Operating systems in the virtual environment.
- **Applications**
- **Data**

Typ cloudovej služby	Zodpovednosť			
	Infraštruktúra	OS	Aplikácia	Dáta
SaaS	CSP	CSP	CSP	Tenant
PaaS	CSP	CSP	Tenant	Tenant
IaaS	CSP	Tenant	Tenant	Tenant

Figure 4.2 Overview of responsibility in CC services [46]

From the figure above, we can see that the *tenant* bears a large degree of responsibility with an IaaS service, whereas with SaaS-type services the provider bears greater responsibility. We can conclude from this that, in the monitoring system of a CC platform, the CSP needs to monitor That part for which it is responsible; the rest falls within the competence of the individual clients [46].

4.4 Presenting the collected data to the stakeholders

As we have already discussed in chapter 4.3, the stakeholders include the provider and the users (clients). In our view, the ways in which data will be presented to them need not differ fundamentally; the difference will only be in the amount and type of data presented. Based on information from Google, we can distinguish two main ways of presenting information from a monitoring system, namely reports and a form of access to monitoring data [47].

4.4.1 Reports

It is not precisely defined what different types and what amount of information must be included in such reports. This depends mainly on the requirements of customers and the capabilities of the CSP. In an effort to identify different types of suitable reports, we can draw inspiration from the technical documentation of the W3500 cloud system from IBM [48]. From the IaaS perspective, we consider reports on machine activity to be significant. These are used to track the physical and virtual resources that are used within the cloud. Within this category, there may be several types of reports:

- **Compute node reports –** Useful for the CSP. May contain usage patterns, averages and potential future trends of CPU and memory utilization. Based on such a report, it would be possible to identify nodes that are either underutilized or overloaded. The following information may be suitable for this type:
  - status,
  - CPU speed (MHz/GHz),
  - average CPU utilization over N days (%),
  - maximum CPU utilization reached over N days (%),
  - memory capacity,
  - average memory utilization over N days (%/GB),
  - maximum memory utilization reached over N days (%/GB), and
  - number of instances running on the compute node.
- **Reports on utilized allocated resources –** Useful for customers. This type of report may contain the same information as the type above; however, the whole will not be taken as the capacity of the compute node but as the capacity allocated to the given customer.
- **Utilization of allocated storage resources –** Useful for customers. A similar type of report may be generated for the CSP, where, however, the whole is taken to be the storage resources on the compute node or across the entire platform. It provides information on storage utilization. This information may include:
  - allocated storage (GB),
  - average storage utilization over N days (GB/%), and
  - maximum storage utilization reached over N days (GB/%).
- **Virtual machine utilization –** Useful for the customer. From this type of report, the customer can obtain information about their VMs. Below we list the information that such a report could contain:
  - VM identifier,
  - status,
  - information on the allocated CPU:
    - number of cores,
    - frequency.
  - Current CPU utilization (%),
  - average CPU utilization over N days (%),
  - allocated memory (GB),
  - allocated storage (GB),
  - current memory/storage utilization (GB/%), and

- average memory/storage utilization over N days (GB/%).

- **Disk information –** Useful for the CSP. Provides the CSP with information on the state and utilization of individual disks. Based on such data, it may be able to identify their load or a deterioration in performance, which may herald an impending device failure. It may contain the following information:
  - status,
  - size (GB),
  - average number of read input/output operations per second (*read IOPS*) over the last N days,
  - average response time of read operations over the last N days (ms),
  - average response time of write operations over the last N days (ms),
  - the compute node in which the disk is located, and
  - if possible, a list of instances that use the disk [48].

4.4.2 Access to monitoring data – plotting metrics

For a better understanding of the behaviour of numerical indicators, it is appropriate to plot them in the right way. In practice, it is common to visualize the most important metrics on a single screen (*dashboard*), which serves as an effective visualization tool for tracking and displaying data from various sources. As a result, it provides an overview of the state and performance of the monitored attribute. How to plot which data is the subject of the field of data analysis. Here we mention several common visualization techniques:

- **Pie chart** – Divides the whole into sections. Provides a simple way to compare the sizes of the individual components within the whole.
- **Line chart** – Suitably displays the change of one or more metrics over time. A suitable way to quickly detect potential anomalies [49].
- **Gauge (*gauge*) –** It can be imagined as a needle speedometer in a vehicle. Together with a numerical indicator, it is suitable for displaying the current value of a metric [50].

Visual communication should be simple, so as to enable the target group to extract the required information from visual interpretations quickly and easily. Recommended practices for creating a useful and clear visualization are therefore given.

- **Assessing the target group –** When creating visualizations, several basic questions need to be considered. The most important are whom the visualization is intended for, what their knowledge of the subject is, and what information they are trying to obtain from it. If possible, we recommend first presenting the visualization to part of the target group to obtain feedback before putting it into production.
- **Choose suitable visualization techniques –** For different types of metrics, the right type of chart needs to be considered. We have already discussed this topic in the section above.
- **Keep it simple –** In general, it is advisable to follow the approach of visualizing as little data as possible while providing the greatest possible informative value [49].

4.4.3 Access to monitoring data – alerting

Alerting on significant events is one of the most critical and most important parts of a monitoring system. From this point of view, it is also the most difficult to define precisely, since every system is different. This means that individual indicators will have different weights in different systems. There are certain practices that monitoring system administrators can follow to create a successful alerting strategy. Microsoft recommends considering the questions below to determine whether a detected symptom is a suitable candidate for creating an alert. This strategy, however, assumes an industrial approach, in which a specific team is assigned to resolve each type of problem (e.g. hardware/software).

- **Is it urgent? –** A problem that has arisen does not necessarily mean that an immediate response from the administrator is needed [51]. It is common practice to define alert levels based on their severity, as shown in **Table 4.1**.
- **Are customers affected? –** Are the users of the service affected by the problem that has arisen? If so, it is advisable to escalate the problem to urgent and inform the customer that the problem is being resolved.
- **Are dependent systems affected? –** Can the problem that has arisen create further alerts from dependent systems [51]? For example, an outage of a compute node may give rise to subsequent alerts about non-functionality, for example of the *OpenStack* component Nova. In a multi-team environment, it is important in such an event to ensure that several teams do not work on the same problem.

Level	Level name	Description
Severity 0	Critical	Outage of a critical service element, loss of availability, significantly reduced performance. Requires immediate action.
Severity 1	Error	Reduced performance, loss of availability of part of the service. Requires attention, but not immediately.
Severity 2	Warning	A problem that does not involve reduced performance or loss of availability but has the potential to cause them if it escalates.
Severity 3	Informational	Does not indicate a problem but provides interesting information to the operator.
Severity 4	Verbose	Does not indicate a problem but provides detailed information to the operator.

Table 4.1 Alert levels [52]

5 SURVEY OF SUITABLE TOOLS FOR THE SOLUTION

From the work carried out so far, it is clear that the tool we are looking for should be able to collect metrics from physical servers, network devices and hypervisors. It should also be able to obtain information directly from the *OpenStack* environment. It should offer evaluation and visual presentation of the collected data, at least to the platform administrators. We also attach importance to the ability to create rules on the basis of which administrators will be alerted when set values are exceeded. Support for multi-tenant architecture would also be an advantage, allowing users to access monitoring data from their virtual environments.

There are a large number of tools on the market, intended either for generic system monitoring or for specialized cloud monitoring. Most of these systems are paid; we will, however, focus on those with an open licence. Below we give a brief overview of some of them, and further on we will primarily focus on comparing the *Zabbix* solution and the *Prometheus* solution.

5.1 General overview of tools

5.1.1 Nagios

In the world of open-licence monitoring solutions, the *Nagios* service is frequently mentioned. The developers of this solution offer *Nagios Core* under an open licence, which we can regard as the core of the service, providing a tool for infrastructure monitoring, alerting administrators and a large number of free add-ons created by the developers and the community. Thanks to these, it is also possible to monitor services running in *OpenStack*. *Nagios* is, however, older software, which shows in its user interface. In the free version, multi-tenant architecture is not natively supported. In summary, with this solution we would be able to monitor the state of the physical, network and virtual infrastructure, as well as the services within *OpenStack*. Visualization and alerting are also natively supported [3], [37], [53].

5.1.2 Graphite

Graphite is an open-licence monitoring tool characterized by storing data in the form of time series. It works as a central point to which users can send metrics in a defined format, collected using scripts or other third-party solutions, most often, however, *collectd*. This solution provides overall visibility over the infrastructure and supports both visualization and alerting of administrators. Obtaining data directly from *OpenStack* could be problematic.

For visualization, it can be connected to the *Grafana* tool, which supports multi-tenant architecture in a certain form. This solution is highly customizable, but its administration requires in-depth knowledge of the tool [54], [55].

5.1.3 Collectd

This daemon, see also [56], can collect both system and application metrics and provides mechanisms for storing them or makes them available over the network. Its strengths are speed, since it was written in C and optimized for performance, and the wide availability of various plug-ins. This tool is under continuous development and well documented. It does, however, have its limitations. On its own, it cannot visualize the collected data, as it has no graphical user interface. Alerting capability is supported, but it is significantly limited to simple checking of preset values [56]. We see a big plus in the possibility of connecting it with solutions such as *Graphite* or *Prometheus*, since *Collecd* provides plug-ins that convert the collected data into the required format and then make them available for collection.

5.1.4 Grafana

Grafana is an interactive data visualization platform. Although it cannot perform monitoring on its own, in combination with monitoring solutions it provides a powerful tool for both visualization and alerting. It can work with data from various sources such as *InfluxDB*, *Prometheus*, *Graphite*, etc. It is offered in three variants:

- **Open licence** – It is free of charge and offers all the basic functionality.
- **Cloud solution** – Various plans are available; the free plan includes ten thousand metric time series in *Prometheus* format, fifty GB of log storage, etc. The official website presents it as the simplest solution for beginners.
- **For businesses** – A paid solution with full support and additional functionality, such as the ability to send reports, secure separation of clients, etc. [57].

Also worth mentioning is the strong connection between the *Prometheus* solution for data collection and *Grafana* for its visualization.

5.2 Zabbix

Zabbix presents itself as a distributed, open-licence monitoring solution. It can monitor a large set of parameters, from network devices through physical servers, VMs, applications, services, etc. It has built-in support for SNMP, which makes it an excellent tool

for monitoring network infrastructure. Data collection is performed either directly by the server, by a proxy server or by means of agents on end devices. *Zabbix* has good support for configuring alerts. Different evaluation rules for sending an alert can be set for different types of metrics. Built-in visualization tools make it easy to visualize the collected data, which are stored in an internal database. The community has so far created a large number of templates for various types of monitoring, e.g. templates for monitoring Cisco devices via SNMP.

The *Zabbix* architecture consists of the following components:

- **Server** – The central component where all configurations and data are stored.
- **Database storage** – *Zabbix* stores everything in database storage.
- **Web interface** – An interface that allows users to interact with the system. It is usually deployed on the same device as the central server.
- **Proxy** – Can collect monitoring data instead of the central server. It is not a mandatory part, but it is suitable for a distributed deployment.
- **Agent** – Used to monitor local resources on end devices [58].

To achieve a multi-tenant architecture, connection with the *Grafana* solution is recommended [59].

For monitoring the physical infrastructure, *Zabbix* appears to be the most suitable solution. However, for monitoring *OpenStack* services, as well as VMs through communication with the hypervisor, it would be necessary to verify the functionality of the scripts created by the community for these purposes, or to create our own.

5.3 *Prometheus*

Behind the name *Prometheus* lies a toolkit for system monitoring and alerting. It is part of the *Cloud Native Computing Foundation*, which brings together open-licence cloud-native projects. It stores the collected data as time series, which means that each metric is stored with a timestamp and optional key-value pairs, also referred to as labels. The strengths of this tool include precisely the way the collected metrics are stored, as well as its own query language, *PromQL*, which makes it possible to make efficient use of the matrix-based data storage. We also consider the data acquisition process to be an advantage, as it uses a *pull* model in combination with the Hypertext Transfer Protocol (HTTP). *Prometheus* was designed primarily for reliability, since each server component operates independently and does not depend on any external services. Running it does not require building a large infrastructure. Data collection is performed using so-called

exporters, which can be deployed either on the end station or on another station, depending on the type and method of data acquisition [60]. The architecture of the solution is shown in **Figure 5.1**.

5.4 Summary

For practical deployment, we selected a solution built on the *Prometheus* toolkit and the *Grafana* visualization platform. Our choice was motivated by the progressiveness of these solutions and their strong mutual integration. We were also inspired by the work [34], in which the authors successfully monitored the state of the *OpenStack* platform as well as the hardware components on the physical nodes. However, the authors did not address the

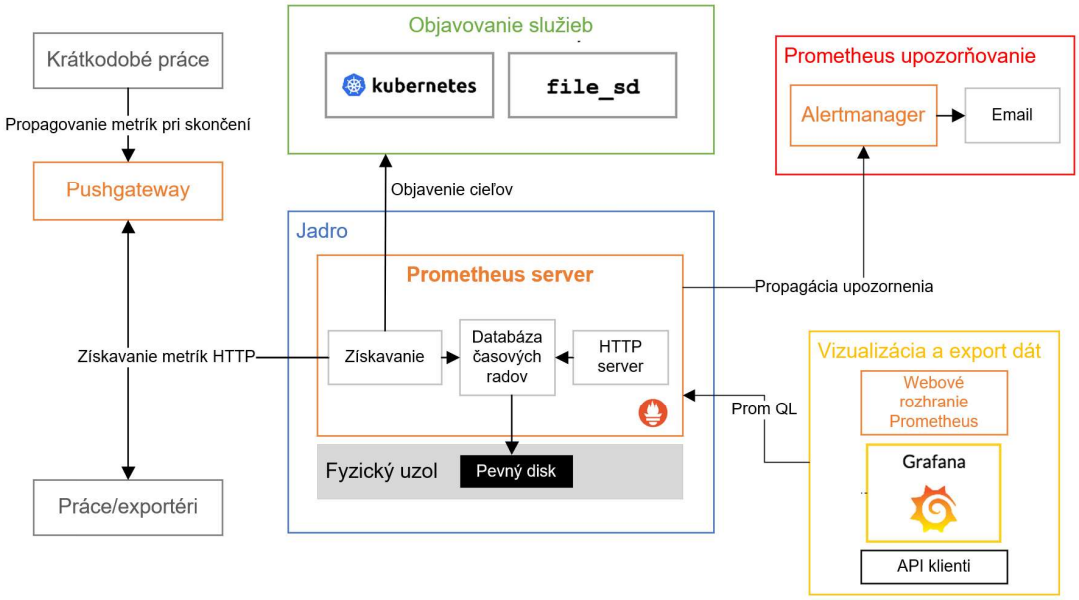


Figure 5.1 Prometheus architecture [60]

monitoring of VMs inside the cloud, nor of network devices. Likewise, they did not place great emphasis on visualization and used preconfigured templates. In our solution, the aim will be to improve the monitoring system with respect to these aspects as well.

6 DESIGN OF THE TECHNICAL SOLUTION

In this chapter, we will address the design and deployment of the monitoring system itself. First, we will define the requirements for the solution, then we will attempt to identify the hardware requirements for such a system and select the technologies needed for its implementation. Subsequently, we will present the proposed architecture and describe its deployment.

6.1 Requirements for the solution

As discussed in Chapter 1.2, there are several motivations for using a monitoring system. The main objective is to design and deploy a monitoring system for a CC platform built on the *OpenStack* solution. From our point of view, the following processes are important for the owner.

Capacity and resource planning, and **capacity and resource management,** since a physical infrastructure will be operated whose health, smooth running and sufficient computing resources are critical for the functionality of the entire system.

Data centre management need not be given such significance, since the size and complexity of the infrastructure will not reach the dimensions of an industrial data centre in the foreseeable future. However, it will be advisable to consider the performance of the monitoring system in the event of the infrastructure being scaled in the future.

We attach a higher degree of importance to **SLA management**, since it is anticipated that CC services will be provided to further customers.

Billing is not the subject of this work.

Problem detection is the key reason for operating a monitoring system; therefore, our solution should be clear and easy to follow. At the same time, it should enable rapid detection of a problem as well as alert administrators that a problem has occurred.

In **performance management**, the system should be able to identify a potential decrease in performance on individual nodes and inform at least the administrators about it.

Within **security**, we do not envisage monitoring customer data in order to guarantee its security.

We therefore require the solution to:

1. **Be able to monitor the availability and performance** of all hardware elements that serve to run the platform. We include among these elements network and computing devices, i.e. servers. With regard to performance, we have identified the three most critical components, namely CPU, RAM and storage. In the case of network devices, interfaces can also be included here. The deployed system must be able to collect metrics containing such information that the utilization of the aforementioned components can be determined from them, with the critical limits of the system identified. Moreover, in addition to the hardware resources, it is advisable for the solution to monitor the amount of *OpenStack* platform resources used. We include among such resources vCPU, RAM and storage from the platform's point of view.
2. **Be capable of monitoring the state of the provided services**. Within the scope of the thesis assignment, we address the IaaS service, which on the target platform for which we are designing the system consists of six separate services: *nova*, *cinder*, *heat*, *keystone*, *neutron* and *glance*. In a potential SLA, it would be worth considering listing these services separately, since each manages a certain aspect of the whole, as stated in Chapter 4.2. In this context, we require the system to be able to monitor, store and evaluate their availability within a specified time window, e.g. one day, and, as part of monitoring SLA parameters, to collect information about the clients' environments. We consider *OpenStack* projects to be such environments. Within them, the system must obtain data on the size of the available/used amount of memory and storage, and the used/maximum number of instances and processor cores. In addition, it is necessary to monitor CPU and RAM utilization by individual instances. The other metrics listed in Table 2.1 may also be suitable.
3. **Visualize the collected data appropriately**, so that, should a problem arise, it is easier for the administrator to identify its origin.
4. **Reliably and promptly alert the relevant persons** in the event of a problem, for example the system administrators. In general, the provider has no interest in alerting clients to events in their environments, since it may thereby forgo potential profit. However, it may offer such a service as an extra.

6.2 Selection of technologies and hardware requirements

In this chapter, we will describe the selected technologies *Prometheus* and *Grafana*, together with the necessary components, while attempting to define their hardware requirements.

6.2.1 Prometheus server

We will deploy the *Prometheus* server as the core of the entire solution. It will be responsible for collecting, storing and providing metrics to other components. It consists of three basic elements, as shown in **Figure 5.1**. The data retrieval component accesses HTTP endpoints and obtains metrics from them. It then passes them to the storage component, which takes care of storing them efficiently. The last component is the HTTP server, which accepts queries in the *PromQL* format and provides the requested data. This server also functions as a web user interface, which offers basic visualization and a textual overview of the collected metrics, together with the status of the configured endpoints.

Determining the system resources required for smooth operation is not a simple task. This depends mainly on the frequency of collection and the size of the metrics, as well as on the number and complexity of queries to the *Prometheus* server. For a *Prometheus* + *Grafana* environment, the following is commonly recommended for an initial deployment: at least two processor cores, four GB of main memory and twenty GB of storage. After deployment, it is possible to approximately calculate the amount of RAM needed to run the *Prometheus* server; a calculator with an explanation and description is available online at [61].

6.2.2 Tools for metric collection

Building a metric collection system is a complex problem for which no single solution exists, and therefore our design will also use several components. The hardware requirements of the tools listed below cannot be precisely determined from the available sources. It holds, however, that they are very simple programs that do not perform computationally intensive activities. They should therefore not have a significant impact on the overall performance of the host system.

From the point of view of the requirements for the solution, we can classify the tools used into the areas of availability and performance monitoring, and monitoring of the state of the provided services. We have already discussed the description of these requirements in Chapter 6.1.

6.2.2.1 Availability and performance

In the context of collecting metrics ***from physical nodes***, we can use an exporter for Linux systems called *Node exporter* (a component of the *Prometheus* system). It needs to be installed on every end station from which we wish to extract information. This program will then obtain metrics either on the basis of system calls or by accessing system files in

which the OS kernel stores usage statistics. Such retrieval is carried out with the help of collectors, of which more than fifty are available, each for different types of system statistics. Their complete list is available online in the program's documentation, see [62]. After collection, the exporter converts them into a specific format for *Prometheus* and exposes them via an HTTP endpoint for the *Prometheus* server to read.

For ***network devices***, we need to use the SNMP protocol for data collection. *Prometheus* has no native support for this protocol; therefore, we must deploy a specialized exporter. We will deploy it for the purpose of collecting data via SNMP and subsequently converting it into a comprehensible format, since SNMP uses a hierarchical structure for presenting metrics, whereas *Prometheus* uses an N-tuple structure. This exporter is called *snmp_exporter,* and is one of the officially supported ones, together with the above-mentioned *Node exporter*. It consists of two main components: *snmp_exporter* and *generator*. The latter serves to create the configuration file, while the former is the main component responsible for collecting, converting and presenting the metrics. The recommended deployment is on one or more nodes, and this exporter can be thought of as a proxy for the *Prometheus* server. We will discuss the operation of this exporter in more detail later in this work. For more information, see [63].

6.2.2.2 State of the provided services

For ***collecting information about services and resources*** within the *OpenStack* platform, we can use an exporter called *openstack-exporter*. This community solution, written in *Golang*, can obtain metrics from a running *OpenStack* cloud and expose them to the *Prometheus* server. It collects data via the API endpoints provided by *OpenStack*. The exact placement of this exporter is not defined. For its functionality, however, it needs connectivity to the *OpenStack* API endpoints as well as to the *Prometheus* server. The list of all the information it can collect is fairly extensive and is available in the documentation, see [64]. In addition to the above, there is also an exporter released under the company *Cannonical*. However, it is less well documented and at the time of our research appeared to be outdated; nevertheless, for the reader we provide a link to its *github* repository, see [65].

In order to meet the requirements, we still need to obtain ***information about the instances in the cloud***. The method we chose is the extraction of this data from the hypervisors on the compute nodes. A solution called *collectd* can serve this purpose. This daemon can collect an enormous amount of data from the system on which it is deployed with the help of built-in plugins. In this context, we are interested in the *virt* and *write_prometheus* plugins. The first of these uses the *libvirt* API to collect statistics about

the virtual hosts running in the system, as does *virsh,* which we discussed in Chapter 3.3. The second plugin mentioned takes care of converting the metrics into a suitable format. To guarantee functionality, this daemon must be installed on all compute nodes in the cloud.

6.2.3 Visualization and alerting

Although *Prometheus* allows a limited form of visualization of the collected data, its own documentation recommends using the *Grafana* tool for this purpose. It will allow us to create dashboards in which we will be able to configure panels displaying the most important aspects of the system. Each panel will obtain the necessary information using *PromQL* queries to the *Prometheus* server. It is up to the administrators to decide whether to deploy this tool on a separate server or on the same server as *Prometheus*. We see a major advantage in the possibility of integration with other systems that may be added to this solution in the future, thereby creating a single central presentation node. A further benefit is the possibility of configuring alerts directly in the *Grafana* graphical interface, which we will make use of later in this work.

In the official documentation, the minimum hardware requirements are defined as two hundred and fifty-five MB of main memory and one CPU core. When using functionalities such as alerting, it is recommended to increase the number of CPU cores [66].

6.3 Solution architecture

For the purpose of implementing the project, we used a physical server intended for virtualization, running the *VmWare ESXI 6.7 Update 3* operating system. In this environment, we can deploy VMs as needed. Before designing the architecture itself, it was necessary to determine whether we would aim to separate the individual components into separate VMs, or whether we would choose the path of a centralized solution, in which there would be one central point. Although a centralized solution introduces a single point of failure, we decided precisely on this approach for the sake of simpler administration in the future. Subsequently, we must also take into account the physical infrastructure of the cloud, since we wish to deploy two types of exporters on the compute nodes and we will monitor the network devices using SNMP.

6.3.1 Central server

The central server will be the main element of the entire system. For better clarity, we first present a graphical visualization of its components and their mutual communication, see **Figure 6.1**. The basic component here is the *Prometheus* server, which communicates

with the exporters and collects metrics from them. This communication is depicted by black arrows and uses the HTTP protocol. The server initiates the communication by sending an HTTP GET to localhost and the specific port of the exporter. The path to the metrics file is `/metrics` by default for every exporter. The ports of the deployed exporters are listed in **Table 6.1**. If everything works correctly, the exporter sends a 200 OK response together with the metrics, and the *Prometheus* server stores them in the database. As already mentioned, for visualization we will use the *Grafana* tool, which we will also deploy on the central server. It will communicate directly with the *Prometheus* server using *PromQL* queries, to which the server responds with the requested time series.

Below, we present a figure of the architecture and a list of the components that we will deploy on the central server, with descriptions of their roles and the ports on which they accept connections.

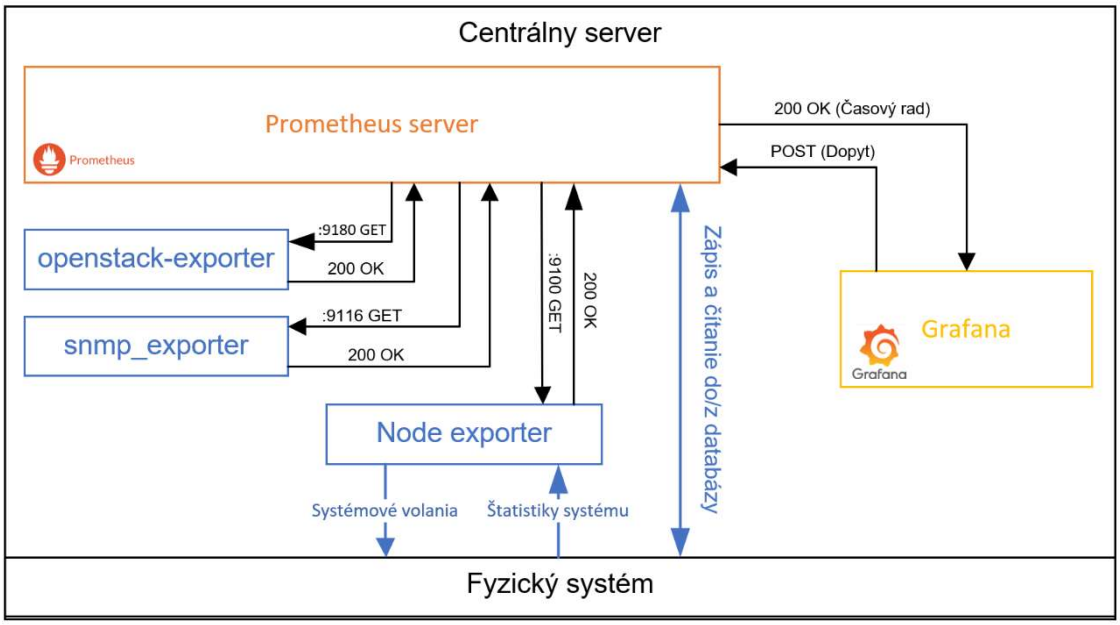


Figure 6.1 Architecture of the central server

Component	Port	Role
Prometheus server	9090	Collecting, storing and providing metrics; provides a web server for user interaction.
Grafana	3000	Visualization and alerting; provides a web server for user interaction.
openstack-exporter	9180	Collection of metrics from the *OpenStack* API
snmp_exporter	9116	Collection of metrics from network devices using SNMP, conversion to the correct format
Node exporter	9100	Collection of system information from the central server

Table 6.1 List of components on the central server

Although ***Openstack-exporter*** and ***snmp_exporter*** will be deployed on the central server, they will collect metrics from the external environment of the cloud platform. We therefore consider it appropriate to clarify how these components work. We will first address the former, whose communication with the API endpoints is represented in **Figure 6.2**. The first step is authentication against the *OpenStack* API, with all configuration data such as the name, password and the address of the authentication endpoint being specified in the *clouds.yaml* configuration file in the specific *os-client-config* format, see [64]. The program then starts sending HTTP GET requests to all available endpoints, thereby obtaining statistics directly from OpenStack. The endpoints do not need to be configured individually; the program discovers them by itself.

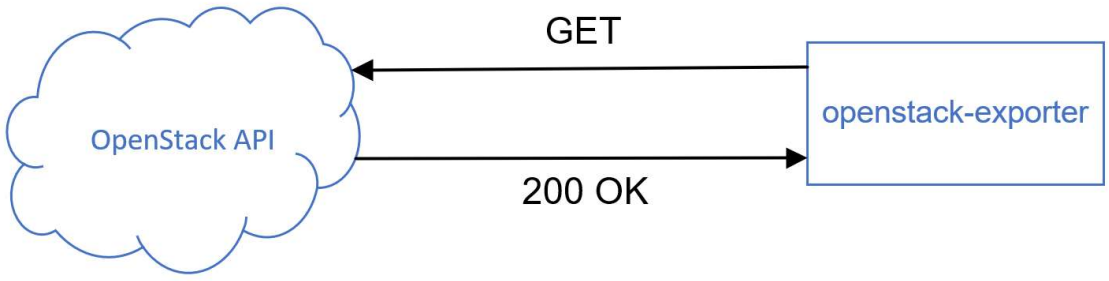


Figure 6.2 Communication of openstack-exporter with the OpenStack API

Snmp_exporter uses SNMP to obtain metrics. This communication is depicted in **Figure 6.3**. All OIDs we are interested in must be specified in the configuration file. This file is generated using the generator. We will discuss this topic in more detail during the deployment itself.

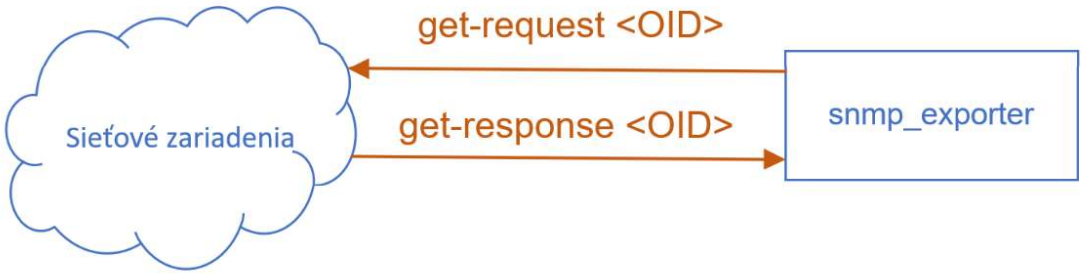


Figure 6.3 Communication of snmp_exporter with network devices

6.3.2 Other components

In addition to the components presented so far, within the architecture we also define monitoring agents on the physical nodes of the cloud platform. As already mentioned in Chapter 6.2.2, on the compute nodes these are the *Node exporter* and the *collectd* daemon with the *virt* and *write_prometheus* plugins. *collectd* is not needed on the controller node. Communication with these agents is identical to that with the exporters on the central server and is presented in **Figure 6.4**. A new exporter appears here, which accepts connections on port 9103.

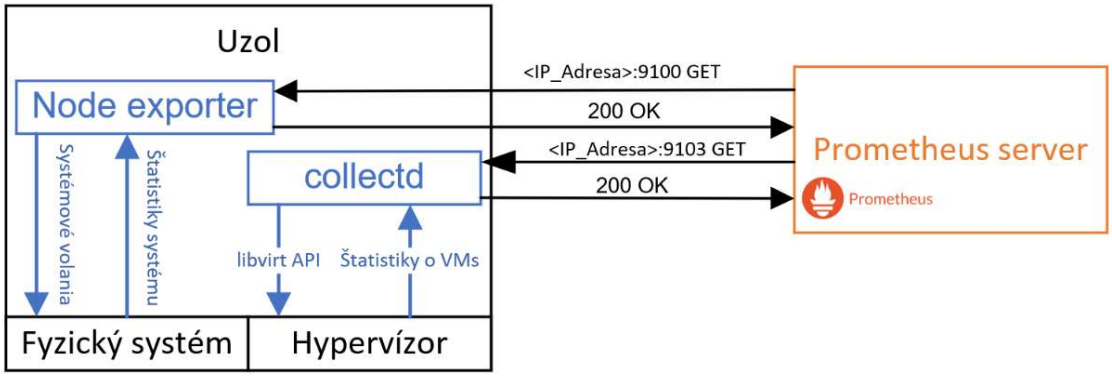


Figure 6.4 Architecture of the other components

6.3.3 Overall architecture

In this section, we connect the system architecture with the physical world. Through consultation with the administrators of the cloud platform, we ascertained the composition of the infrastructure. It consists of two nodes, one being a controller node and the other a compute node. Connectivity between them and to the department network where the test took place is provided by a Cisco Catalyst 2960-X switch, specifically on ports Gi1/0/25 to Gi1/0/30. It is precisely these three devices that we consider to be the physical infrastructure

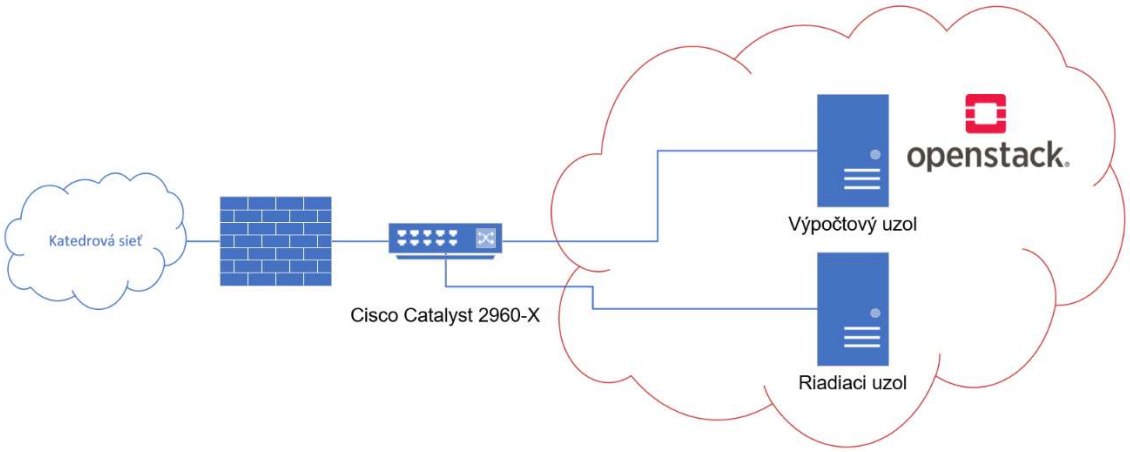


Figure 6.5 Topology of the monitored cloud

that we need to monitor. It is depicted in **Figure 6.5**.

In addition to the platform, we also need to clarify the environment of the monitoring server. It consists of two VMs, one serving as the server itself and the other as a router and basic firewall. The topology of this environment is illustrated in **Figure 6.6**.

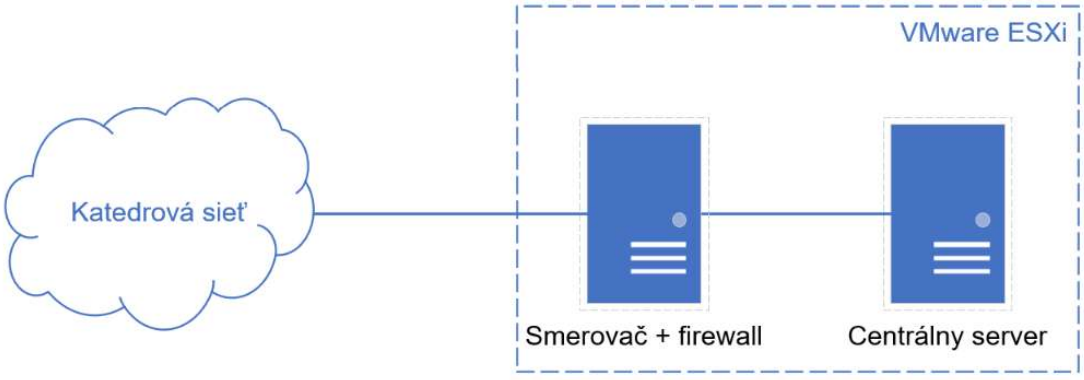


Figure 6.6 Environment of the monitoring server

By combining all the elements together, we obtain a view of the overall architecture of the solution, see **Figure 6.7**.

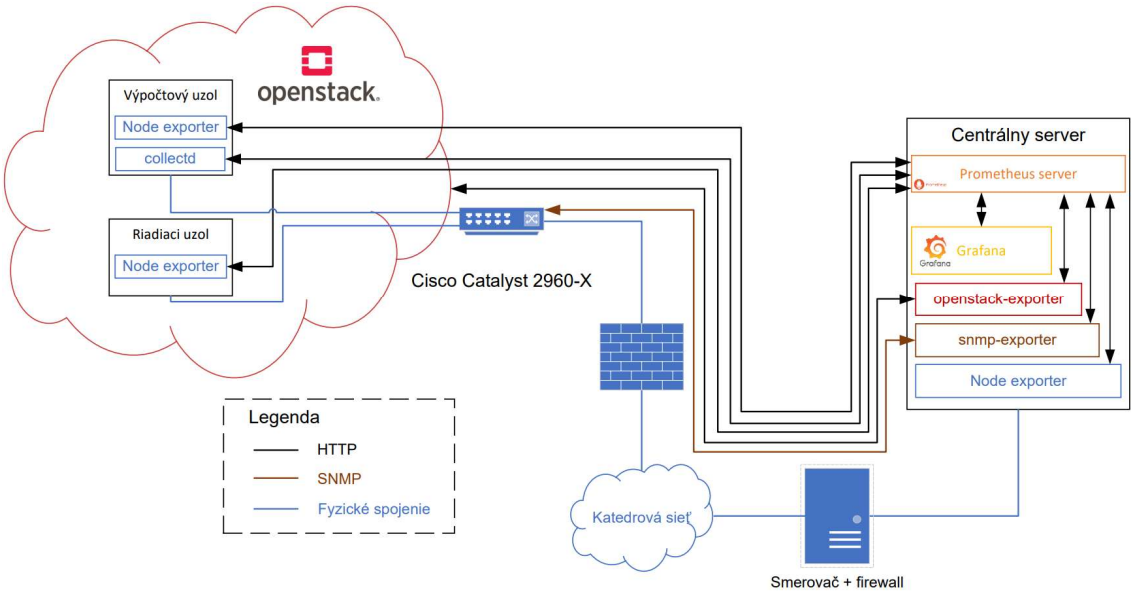


Figure 6.7 Overall architecture

6.4 Deployment

On the basis of the architecture we defined in the previous chapter, this monitoring solution needs to be deployed and brought into a functional state. In this chapter, we will describe the process of deploying the individual components and verify their functionality. We will deploy the components on three different machines. To begin with, it is appropriate to state their specifications; for the nodes in the cloud we are interested only in the OS, whereas for the central server we will also state the allocated computing resources.

- **Controller node:**
  - **OS:** CentOS Linux 7 (Core)
  - **Kernel:** Linux 3.10.0-1127.18.2.el7.x86_64
- **Compute node:**
  - **OS:** CentOS Linux 7 (Core)
  - **Kernel:** Linux 3.10.0-1127.18.2.el7.x86_64
- **Central server:**
  - **OS:** Ubuntu 20.04.5 LTS
  - **Kernel:** Linux 5.4.0-146-generic
  - **CPU:** 2 cores

- **RAM:** 4 GB
- **Storage:** 300 GB

6.4.1 Prometheus server

Since we consider this element to be the core of the entire solution, it will also be the first component deployed. We will install it on the central server. The method of deployment, initial configuration and use is well described in the official documentation, see [60]. Below, we briefly describe our procedure.

The first step is to download the relevant archive. At the time of deployment, the latest version was labelled 2.37.1. To obtain the download link for the *Prometheus* archive, see the documentation, specifically the section *Introduction->First steps*. For downloading, we recommend the *wget* tool. After downloading, the archive needs to be extracted, preferably with the *tar* tool. It is up to the administrator to decide in which directory to store the resulting folder. We used the `/opt` directory, which, according to the Linux file system documentation, should be used for installing all third-party applications. We will later want to install all components into this directory. We named the resulting folder for *Prometheus* `prometheus_server`, and among other things it contains the two most important files: the binary for launching, named `prometheus`, and the configuration file `prometheus.yml`. The configuration file contains a few preconfigured lines that allow the server to be started immediately after unpacking, in which case it monitors itself. Owing to the limited scope of this work, we will not demonstrate this, and if interested, we refer the reader to the official documentation, where the first interaction with the solution is described in detail. As the last step, we recommend setting up the application as a service. This configuration will allow the application to start automatically at system startup and will make interaction with it easier. Since we wish to configure all components of the system in this way, in Chapter 6.4.7 we present a generic procedure together with a reference to the files we used. In our deployment, we use three program arguments.

- **storage.tsdb.path** - To specify the path to the directory where the time series database will be located.
- **config.file** - To specify the path to the configuration file.
- **storage.tsdb.retention.time** - To set the data retention period, which is fifteen days by default; we require a value of thirty days.

6.4.2 Node exporter

We will deploy this exporter on three stations. The advantage is that this process does not differ between the given OSs. A further benefit is the detailed installation guide in the official *Prometheus* documentation, see [60], section *guides->monitoring linux host metrics with the node exporter*.

The procedure is very similar to the deployment of the previous server. First, it is necessary to download the archive, extract it and run it. Naturally, we adhere to the `/opt` installation directory on every device. Again, we recommend configuring this program as a service. In the case of exporters, it is easy to verify that they are functioning correctly, namely by using a uniform resource locator (URL) client. Since the exporters expose metrics on an HTTP endpoint, we can access this content using the aforementioned client, also called *curl*. As mentioned in 6.3.1, *node exporter* runs on port 9100, and the path to the content is `/metrics`. We will use this information when entering the command: `#curl http://localhost:9100/metrics`. As output, we expect a fairly extensive listing of metrics together with their values.

6.4.3 Snmp exporter

In the context of deploying and configuring exporters, we consider this one to be the most demanding to put into operation, mainly because of the process of configuring it using the configuration file generator. As in the previous cases, the first step is to download the program's binary files, in our case to the central server. For this purpose, we refer the reader to the official repository, which also contains documentation describing the installation and the generation of the configuration, see [63]. At the time of deployment, the latest available version was numbered 0.21.0. Into the `/opt` directory, we cloned two new projects from the *github* repository, named `snmp_exporter`, which contains the basic configuration file and the executable binary, and `snmp_exporter_generator`, containing the entire project including the configuration generator. Since generating the configuration file is a prerequisite for correct functionality, we will describe this process first.

Within the downloaded project in the `snmp_exporter_generator` folder, the most important directory is `generator`. It contains two elements: the generator configuration, named `generator.yml`, and the `mibs` directory, into which it is necessary to download all MIB files that either directly contain the OIDs we require, or other MIBs used by us that depend on them. It is precisely the identification of the required OIDs and subsequently of the corresponding MIB files that is a time-consuming task. For searching, working with and downloading MIB files, we recommend the following sources: [67], [68].

Below, we attempt to summarize the metrics we require, their OIDs and the corresponding MIB files.

Required metrics:

- **System uptime**
  - Name: *sysUpTime*
  - OID: 1.3.6.1.2.1.1
  - MIB: SNMPv2-MIB
- **Information about interfaces**, such as name, speed, current traffic, status, etc.
  - Names: *interfaces* and *ifXTable*
  - OID*:* 1.3.6.1.2.1.2, 1.3.6.1.2.1.31.1.1
  - MIB: IF-MIB
- **Processor statistics**
  - Name: *cpmCPU*
  - OID: 1.3.6.1.4.1.9.9.109.1.1
  - MIB: CISCO-PROCESS-MIB
- **Memory information**
  - Names: *ciscoMemoryPoolUsed* and *ciscoMemoryPoolFree*
  - OID: 1.3.6.1.4.1.9.9.48.1.1.1.5 and 1.3.6.1.4.1.9.9.48.1.1.1.6
  - MIB: CISCO-MEMORY-POOL-MIB

The list of all MIB files we downloaded is available in Annex A: List of MIB files for the snmp_*exporter generator*, with some files being downloaded by the generator itself when the program is built and some needing to be downloaded manually.

Since the generator is dynamically bound to the downloaded MIB files, it must be compiled and built when attempting to generate the configuration, as also stated in the documentation on *github* [63]. Before that, it was necessary to modify the generator configuration, on the basis of which it creates the configuration file for the exporter.

After downloading, we have the generator configuration file `generator.yml`, partially preconfigured. In our case, we needed to add the use of SNMP version two, the definition of the *community string* for authentication and the addition of a definition for collecting processor and memory statistics. For simplicity, we made these changes in the configuration of the module named *if_mib*, which is preconfigured to collect information about interfaces. Its final content is available in **Chyba! Nenašiel sa žiaden zdroj odkazov.**, configuration file.

Since this tool was programmed in the Go language, it is necessary to additionally install this language in the OS. An installation package is available for *Ubuntu,* but unfortunately at the time of deployment it contained Go version 1.13, whereas compiling the generator requires at least version 1.19. We therefore had to perform the installation manually, using the instructions in the official Go documentation, available at [69].

As the last step, we built the generator using the command `#make generator mibs` and created the configuration for the exporter with the command `#make generate`, which generated the `snmp.yml` file, which we moved to the `snmp_exporter` directory, where the exporter's executable file is located. As in the previous cases, we set up the application as a service and launched the program.

For verification, the exporter provides a user interface available on port 9116. Here, we can verify metric collection by entering the IP address of the target device into the *Target* field and pressing the *Submit* button. **Figure 6.8** shows our attempt to obtain metrics from the 2960X switch, see **Figure 6.5**.



Figure 6.8 snmp exporter interface

In our case, we verified the functionality, as demonstrated by the output shown in the figure below.

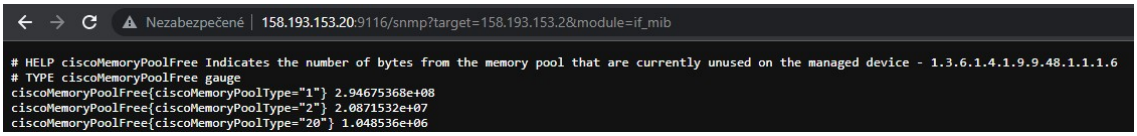


Figure 6.9 Beginning of the listing of metrics obtained using *snmp exporter*

6.4.4 Openstack exporter

The documentation lists as many as four different ways of deploying this exporter. For simplicity, we chose installation using the available *snap* package. The installation must

be performed as the privileged *root* user, and its procedure comprises a single command, given below, which installs the exporter from the *edge* channel and gives us access to the latest functionalities.

```
#snap install –channel edge golang-openstack-exporter
```

Operating the application is well described in the documentation; we therefore refer the reader to [64].

As part of the deployment, we consider it appropriate to mention the problems we encountered. The first concerns the location of the `clouds.yaml` configuration file, which is described in the documentation. We recommend placing this file in the `/var` directory, since, based on our experience, with any other location access to the file is blocked by the *apparmour* application. To define the path to the file, we used the program argument *os-client-config.* In this work, we will not present the content of our file, since it contains private information concerning access to the administrative parts of the cloud.

We verified the functionality of the exporter in the same way as in 6.4.2, but using port 9180.

6.4.5 Collectd

We wish to install the *collectd* daemon only on the compute node. For this purpose, an installation package is available for *collectd* as well as for the required *virt* and *write_prometheus* plugins, but only in the EPEL repository, which is a repository intended for the *RedHat* OS. In the *CentOS* OS, this repository can be enabled. We present the installation process below:

```
//installing the epel repository
#yum install epel-release
//installing collectd with plugins from the epel repository
# yum --enablerepo=epel install collectd collectd-virt collectd-
write_prometheus
//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)
```

In this case, the application has already been created as a service; to have it start automatically at system startup, we only need to enable it using `#systemctl enable collectd`.

After installation, it is necessary to make changes to the configuration file, which by default has most of its configuration in the form of comments beginning with the "#" character. This subsequently makes changes easier, since it is often sufficient to delete this character from the beginning of a line to activate the given configuration. The path to this file in our installation was as follows: `/etc/collectd.conf`. For our purposes, we need to remove the comment-indicating characters from the lines containing *LoadPlugin virt* and *LoadPlugin write_prometheus* in order to load the plugins. Furthermore, we need to modify the content between `<Plugin virt></Plugin>` and `<Plugin write_prometheus></Plugin>`. For descriptions of the meaning of the individual lines, we recommend the official manual, which is described in detail, see [70]. Our configuration is available in **Chyba! Nenašiel sa žiaden zdroj odkazov.**, configuration file.

We again verified the functionality using the `#curl` command, with *collectd* exposing metrics on port 9103. We also verified that the service was running by displaying its status. Here, we encountered an error message: `collectd[26847]: virt plugin: Array index out of bounds: tag_index = 10`. In an effort to trace the reason for this message, we found the following: since CentOS7 is already an older operating system, it is not possible to install the latest version of this daemon on it, owing to dependencies on libraries that are not available for this OS. The latest supported version is 5.9, whereas the latest version overall is 5.12. This is partly reflected in the amount of data we are able to extract from individual VMs, since version 5.12 uses a newer version of the *libvirt* API, which provides additional functionalities. It is also the origin of the aforementioned error message, which has been fixed in newer versions. From our observation, this error has no impact on the correct running of the daemon.

6.4.6 Grafana

The installation process is described in detail in the official documentation, see [66], section *Setup->Install Grafana.* In our deployment, we used installation via a `.deb` package. With this form, any updates must be performed manually. Between the time of deployment and the writing of this work, the developers created an APT repository that can be added to the *apt* package manager on the host system. The process of installing and updating the application is then automated.

After installation, we can access the *Grafana* web interface on port 3000, see **Figure 6.10**. For the first login, we use *admin* as both the username and the password. We were then prompted to change the administrator password.

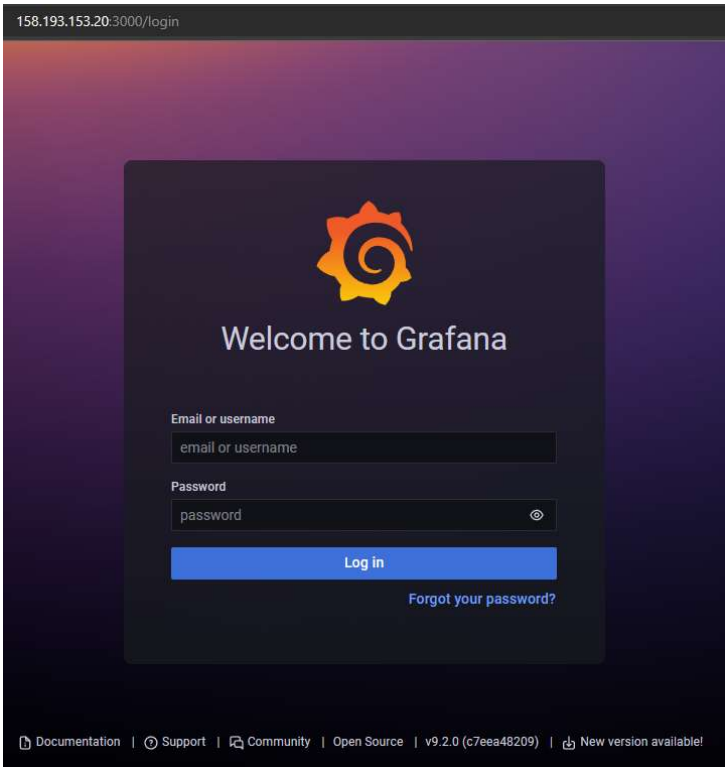


Figure 6.10 Grafana web interface

6.4.7 Application as a service

If we need the application to be active automatically immediately after system startup, we can set it up as a service and enable it to start automatically. We recommend performing this process as the privileged *root* user. Before the configuration itself, we need to have a text editor of our own choice available, in our case *nano*, and to know the path to the application's executable file.

The first step is to create a configuration file for our new service, which must be located in the `/etc/systemd/system` directory. Its name must comply with the following convention: `<SERVICE_NAME>.service`. If we wanted to create *Prometheus* as a service, the configuration file could be named `prometheus.service`. Next, we need to edit this file appropriately. There is a large number of configuration options, but within our deployment we will use only some of them. If interested, we refer the reader to [71], where the whole topic of this type of file is described in a comprehensible manner. To create the

configuration file, see the template or the configuration file we used for the given program, which is likewise available in the appendices to this work.

After completing the configuration file, it is still necessary to:

1. Instruct the service manager to reload the configurations using: `#systemctl deamon reload`.
2. Enable automatic starting of the service at system startup using: `#systemctl enable <SERVICE_NAME>`.

After performing the above steps, the service manager will attempt to start the application each time the system starts. In addition, we can interact with the program using standard commands, for example: `#systemctl [start|stop|restart| status]`. In the event of problems, we recommend consulting the system journal, which often provides the reason for the malfunction. We can access it using: `#journalctl - xe`.

6.4.8 *Prometheus* configuration and verification

By deploying the exporters, we gained the ability to extract metrics from the areas of interest. However, we need to store them and make them available to other elements of the system, which is precisely what *Prometheus* serves us for. By configuring it, we define from which locations and how often it is to collect metrics. For completeness, we first present a brief overview of the deployed exporters with their locations, see **Table 6.2**.

Exportér	Spustiteľný súbor	Konfiguračný súbor	URL koncového bodu s metrikami (z pohľadu Prometheus)
Centrálny server (158.193.153.20)			
openstack-exporter	/snap/golang-openstack-exporter/103/bin/openstack-exporter	/var/prometheusProject/clouds.yaml	http://localhost:9180/metrics
snmp-exporter	/opt/snmp_exporter/snmp_exporter	/opt/snmp_exporter/snmp.yml	http://localhost:9116/snmp
Node exporter	/opt/node_exporter/node_exporter	X	http://localhost:9100/metrics
Výpočtový uzol (192.168.1.21)			
Node exporter	/opt/node_exporter/node_exporter	X	http://192.168.1.21:9100/metrics
collectd	/usr/sbin/collectd	/etc/collectd.conf	http://192.168.1.21:9103/metrics
Riadiaci uzol (192.168.1.3)			
Node exporter	/opt/node_exporter/node_exporter	X	http://192.168.1.3:9100/metrics

Table 6.2 List and location of the deployed exporters

The syntax and form of the configuration are given in the official documentation [60], section *Prometheus->Configuration*; therefore, we will not describe it in detail in this work.

From the point of view of metric collection, so-called *jobs* are defined in the configuration, which we translate as "práce" (jobs). A job represents a collection of endpoints with the same purpose, e.g. the endpoints of the compute nodes. All metrics collected within such a definition are assigned a special label, which facilitates queries and searching for specific metrics. Within this configuration, various parameters can be defined, such as the collection frequency. The most important definition, however, is the targets, which define the specific endpoints of the exporters. In our case, we configured the following jobs:

- **Prometheus_VM** – Intended for metrics relating to the central server.
- **openstack_exporter_metrics** – Metrics from the CC OS OpenStack.
- **Controller_nodes** – A job intended for metrics from all controller nodes. Although the monitored cloud currently has only one such node, we try to bear scaling in mind. If another node is added, its endpoint is simply added to the targets of this definition.
- **Compute_nodes** – The same as the previous one, but for compute nodes.
- **All_VMs** – A job intended for metrics obtained from virtual instances.
- **network** – Dedicated to metrics from network devices.

We attach the complete configuration in the appendices, see: configuration file. Globally, we set the metric collection interval to fifteen seconds. In the case of the exporter for *OpenStack*, we overrode this value to one minute. We were led to this step by the later finding that the extraction of metrics by the exporter takes twenty or more seconds in some cases. These values can be viewed for all exporters in the *Prometheus* web interface, in the S*tatus->Targets* section, under the *Last Scrape* item, see e.g. **Figure 6.11**.

It goes without saying that after performing the configuration we must verify its functionality, which we do by accessing the web interface of the *Prometheus* server and navigating to the S*tatus->Targets* section. This is shown in **Figure 6.11**. In the figure, we can see the configured jobs, with all their targets indicating a healthy status with the word *up*. For the compute nodes job and *openstack-exporter*, we also displayed the details, where we can see the targets, the status, which labels are added to the metrics from these targets, when the last collection was performed and how long it took, or an error message, if any.

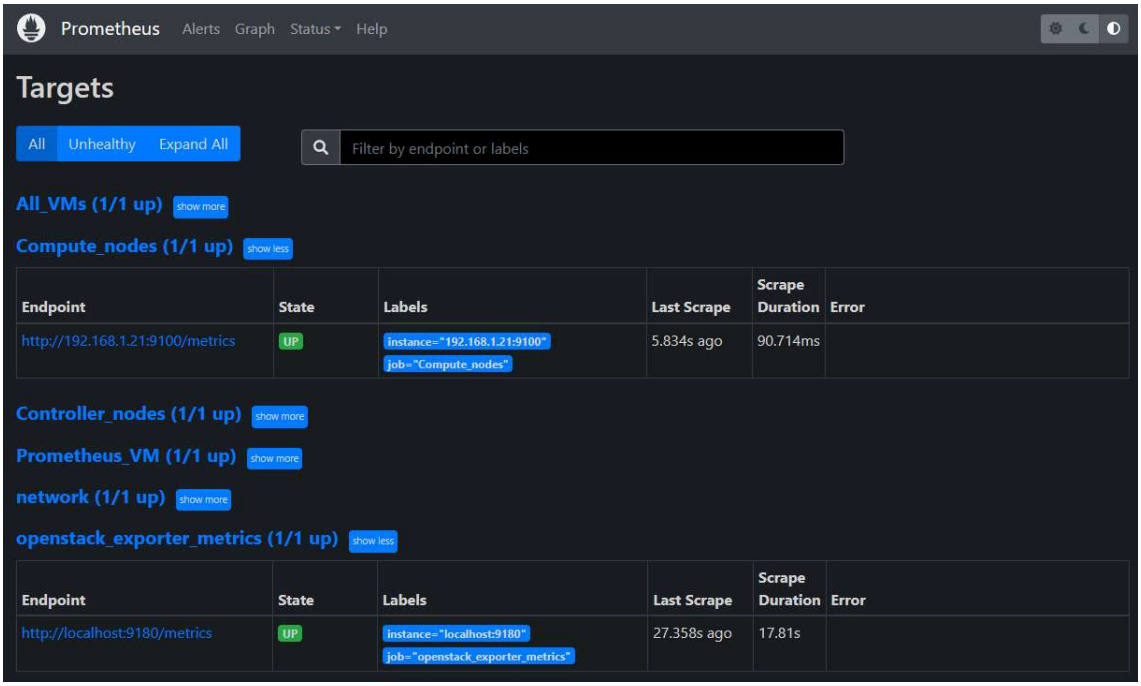


Figure 6.11 *Prometheus* interface - jobs

With the verification of this part, we consider the objective of measuring the technical parameters of the cloud and the network infrastructure for the purpose of gaining an overview of their state to have been fulfilled.

6.5 Preparation for visualization

Collected metrics with a search capability will provide little added value to the users of the monitoring system. In order for the administrator to have a comprehensive overview of the state of the services and the platform, this data needs to be visualized. For the purposes of rapid response to a problem, we in turn need to set up an adequate alerting process. It is precisely these problems that we will address in the further course of this work, guided by two perspectives, as discussed in Chapter 1.3. It follows that we wish to create at least two types of dashboards, which will serve the user and the administrator, while adhering to the recommendations defined in Chapter 4.4.2.

In the case of the user dashboard, the aim will be to visualize the user's available resources and their utilization, together with the metrics listed in Table 2.1.

In the case of the administrator dashboard, the aim will be to display on a single panel the overall state of the services described in Chapter 4.2 and the state of the physical infrastructure, focusing on the elements mentioned in Chapter 3.3.

We will address the topic of alerting only in the context of administrators.

In Chapter 4.4.1, we discussed the suitability of sending reports and their content. Grafana provides such functionality, but only in the paid *enterprise* version.

6.5.1 Connecting Prometheus and Grafana

The first step is the necessity of connecting the monitoring server with the visualization platform. This process begins by logging in to *Grafana*, where we then need to navigate in the navigation panel on the left-hand side to the *Configuration->Data sources* section, where we select the *Add data source* option, see **Figure 6.12 Grafana – adding a**

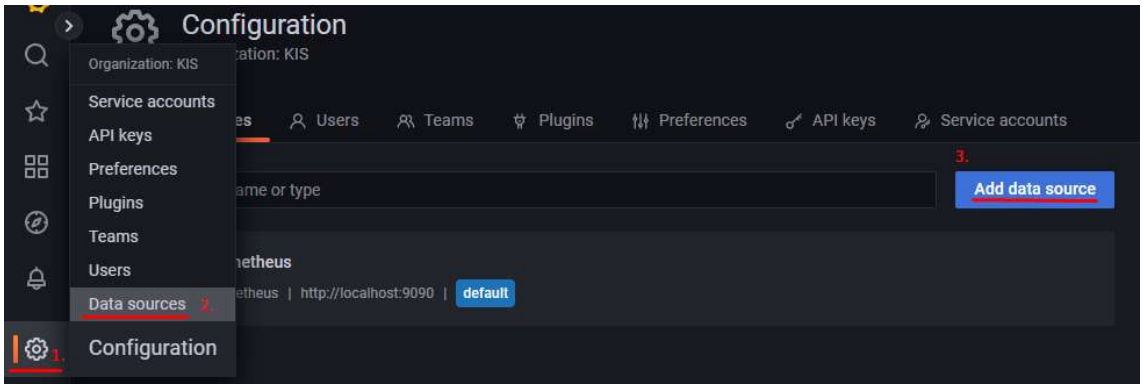


Figure 6.12 Grafana – adding a data source

data sourceFigure 6.12.

A menu of all supported data sources opens; we select *Prometheus* and then fill in the necessary fields. In our case, we named this source *KIS-OpenStack-Old* and set the URL of the *Prometheus* server to http://localhost:9090. At the bottom of the configuration screen, we confirm the entered parameters with the *Save & test* button. Once this is complete, we can proceed to creating dashboards.

6.5.2 Importing preconfigured templates

To begin working with the platform, we recommend importing the available preconfigured dashboards for the exporters. Although we will create our own specialized dashboards corresponding to the requirements of the solution, these generic ones provide a good starting point for further work, since they were created by system administrators and provide an insight into the important metrics from the exporters. In our case, templates are available for *Node exporter*, *openstack-exporter* and *snmp exporter*. All available dashboards can be browsed online at: https://grafana.com/grafana/dashboards/. The method of importing them is very simple, and we only need to know the ID of the required template. It is described step by step in [66], section *Dashboards->Manage dashboards*, paragraph *Import a dasboard*. The process is intuitive; in short, we need to navigate in the

navigation panel to *Dashboards->Import*, enter the ID, load it, select the data source and confirm the operation with the *Import* button. *Grafana* automatically redirects us to the newly created dashboard. These templates are often overly complex, but we recommend them for becoming familiar with the platform.

6.5.3 Folders

Before we move on to creating dashboards, we will explain the concept of folders. *Grafana* provides folders for the logical separation of dashboards and alerts, the default one being named *General*, which cannot be removed. Permissions for access to their content can be set for folders. The topic of permissions, user roles and teams in *Grafana* is fairly extensive, so we refer the reader to the official documentation [66], section *Administration-> Roles and Permisions.* What is important is that each user is assigned one of three available roles: *Admin, Editor* or *Viewer*, on the basis of which we control the permissions for access to and modification of folder content. Within our solution, we created three new folders for dashboards:

- **Old-OpenStack** – For dashboards and alerts intended for administrators, relating to the cloud we monitor. Access to the folder only for users with the *Admin* and *Editor* roles.
- **Templates** – Intended for imported dashboards. Access allowed for the *Admin* and *Editor* roles.
- **Users** – A place for dashboards intended for clients. Access allowed also for the *Viewer* role.

6.5.4 Creating users

Before creating the dashboard for clients, we still need to explain the topic of creating such users. To begin with, we mention that *Grafana* supports the integration of various authentication systems, with all information available in [66], section *Setup->Configure security->Configure authentication*. Within our solution, we did not address such integration. We create and manage users manually in the graphical interface, see [66] section *Administration->User management*, or via the API, see [66] section *Developers->HTTP API->Admin HTTP API*.

For clients, we propose creating an account with a login name that corresponds to the name of the client's *OpenStack* project. Later, when creating the dashboard for clients, this will allow us to use the global *Grafana* variable `${__user.login}`, by means of which we ensure that the given user will have access only to information from their own project.

We likewise assume that the client account will have the lowest possible role, *viewer*, and access only to the *Users* folder, as we defined in chapter 6.5.3. A list of all global variables, including descriptions, is available in the documentation [66], section *Dashboards->Variables->Manage variables.*

6.5.5 Creating dashboards and panels

A monitoring dashboard consists of panels that visualize metrics by means of one or more queries to a data source. In our case, this means *PromQL* queries to the *Prometheus* server. A new dashboard can be created, for example, in the navigation panel under *Dashboards->New dashboard*, see **Figure 6.13**.

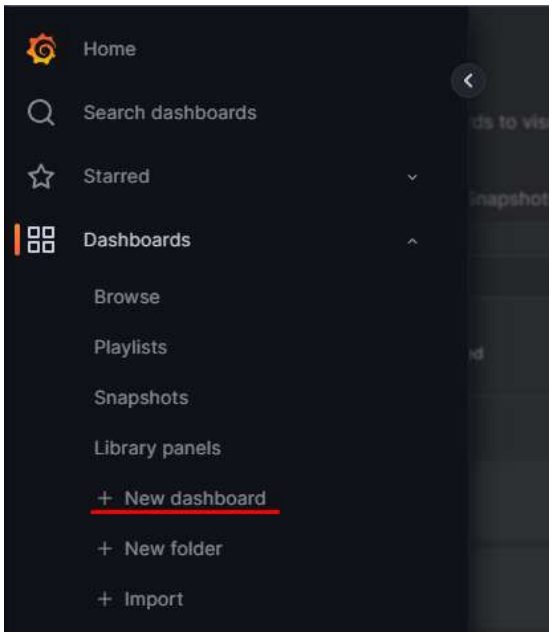


Figure 6.13 *Grafana* dashboard creation – navigation panel

Grafana redirects us to the newly created dashboard, where we can start creating panels. **Figure 6.14** shows the newly created dashboard, with the numbers one to seven marking the individual parts of the interface, whose role we describe below:

1. **Dashboard name** – It should help the user identify what is being visualized.
2. **Add panel** – A button for adding a new panel. It creates on the dashboard the same entity as the one located under the dashboard name in **Figure 6.14**.
3. **Saving** – With every change, the dashboard must be saved in order to preserve it. Saving works by means of versioning, where previous versions are available for import. These are located in the dashboard settings, section *Versions*. When saving the dashboard, we specify its target folder.
4. **Dashboard settings** – This part contains the general settings for the entire dashboard environment. They consist of seven categories, of which we consider the basic settings, e.g. name and variables, to be the most important.
5. **Time range** – In this part, the user can define the time period for which they wish to visualize metrics on the dashboard.

6. **Automatic refresh** – By default, this function is turned off. To dynamically refresh the dashboard content at certain time intervals, this option must be enabled by selecting an interval. Without it, the content is refreshed only when the page is loaded.
7. **Change of display mode** – With this component, the user can switch between three available modes. The first is the basic mode, in the second the navigation panel is hidden, and the third is intended for full-screen display, for example for a dedicated monitor or television.

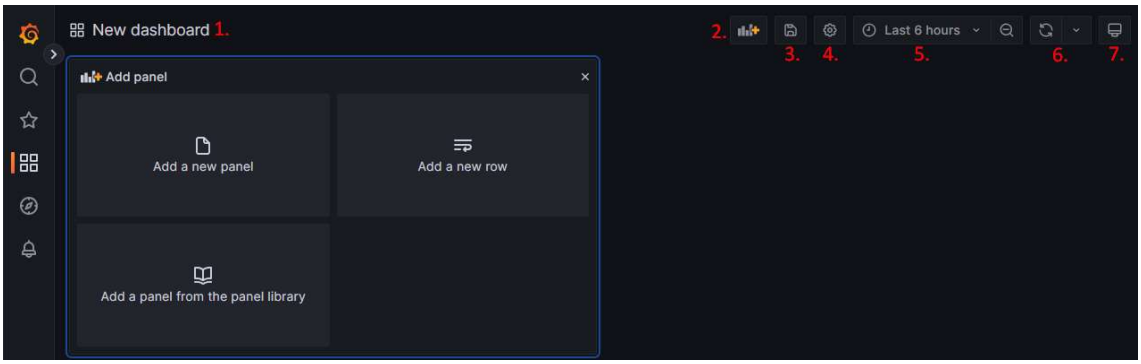


Figure 6.14 Description of the dashboard interface

A large number of visualization options are available within panels, ranging from a simple display of a value, through pie charts and tables, to charts showing the evolution of a metric over time. We will not describe these in detail. More information on creating dashboards and panels can be found in the documentation [66], section *Dashboards->Build dashboards.*

6.6 Monitoring dashboard for clients

In the previous chapter, we discussed the need to create monitoring dashboards, one of which should be intended for clients. We named this dedicated dashboard *Dashboard for users* and saved it in the *Users* folder. When creating it, we aimed for it to provide general information about the environment provided in the cloud, but also detailed information about the user's instances. We will visualize the ratio of available and used resources in the project by means of pie charts. Among these resources we include vCPU cores, the number of instances, RAM and block storage. We also provide the user with a table listing the instances they have created, in which we provide basic information about them, such as name, status, identifier and vCPU utilization, see **Figure 6.15**. To bring our solution closer to the industrial one, we also visualize all the metrics listed in Table 3.1, either for all instances or for the instance selected by the client, see also **Figure 6.16**. The user selects the instance using a drop-down list at the top of the dashboard named *Tenants_instrances,* as also shown in **Figure 6.15**.

We created the individual panels in the *Grafana* graphical interface, but configuration

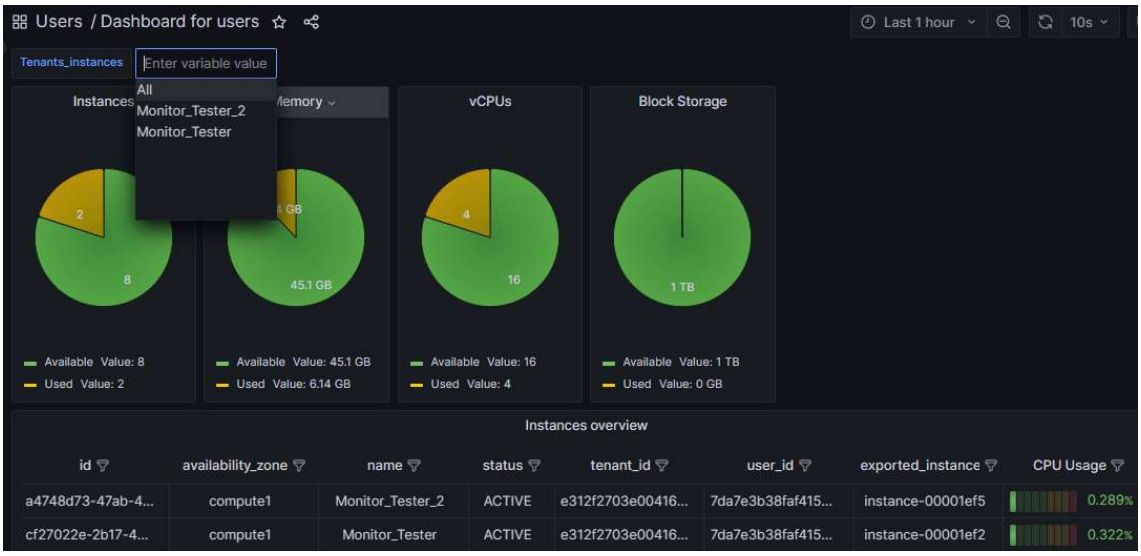


Figure 6.15 Example of part of the dashboard for clients – environment

using the JSON language is also available as a further option. Given the scope of the work, it is not possible for us to discuss the settings of the individual panels. Within their configuration, the most time-consuming task is the correct creation of a set of queries and the correct visual settings. For the reader, we provide the generated JSON model of the entire dashboard, where the queries used can be found under the expression "*expr*". The file is available in the appendices, see **Chyba! Nenašiel sa žiaden zdroj odkazov.**, file JSON model of the dashboard for clients.

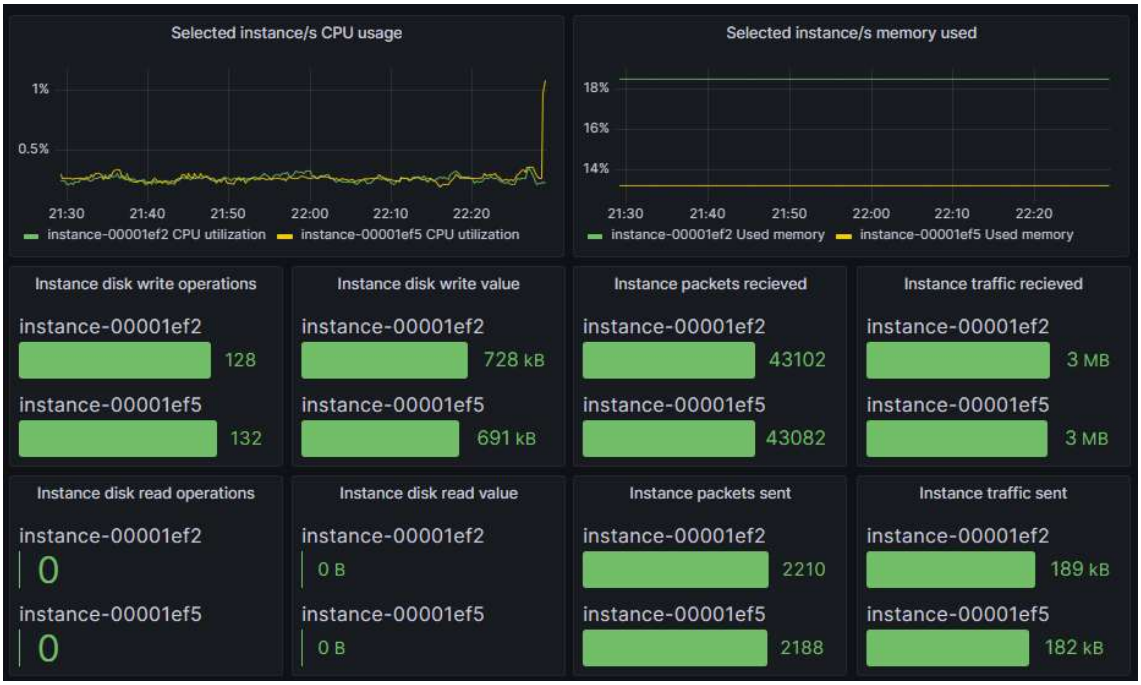


Figure 6.16 Example of part of the dashboard for clients – instance information

An important function of this dashboard is also the dynamic retrieval of data based on the client's login name. By this we mean that, after logging in, each client sees visualized data from their own environment. To achieve this functionality, we used dashboard variables. This concept is fairly extensive and demanding. It is described in detail in the official documentation [66], section *Dashboards->Variables*. The variables we created are of the query type (*query*), and we named them *Tenant* and *Tenants_instances*.

When the dashboard is loaded, the *Tenant* variable executes the query `openstack_identity_project_info{name="${__user.login}"}`, which returns a metric with information about the project whose name matches the content of the global variable `${__user.login}`. According to the documentation, this variable stores the login name of the current user. From the metric, using regular expressions, we extract the name and identifier of the client's project from the labels. This is stored as the value of the *Tenant* variable, which we use in the queries in the visualization panels. As a result, clients are shown only data from their own environments.

The *Tenants_instances* variable uses a query with the nested variable *Tennant* to obtain a list of all the client's VMs. The query we created looks as follows: `openstack_nova_server_status{tenant_id=~"$Tenant"}`. It is precisely on the basis of the value of the *tenant_id* label that we can filter the instances belonging to the client. As with the previous variable, we extract from this list the name and identifier of the

VM, which we store as the value of the variable, which we use in the visualization panels. Screenshots of the settings of these variables are available in the appendices, **Chyba! Nenašiel sa žiaden zdroj odkazov.** – file Panel overview.

In the course of the work, we found that a dashboard created in this way can also be useful for the cloud administrator. If we remove the condition of matching the user name for the *Tenant* variable, such a query returns a list of all projects in the cloud. This creates a drop-down menu with all projects in the dashboard. As a result, the administrator can have a quick overview of all environments and can quickly identify the ownership of an instance exhibiting suspicious activity. We named the dashboard modified in this way *Tenants overview dashboard* and saved it in the *Old-OpenStack* folder. Its JSON model is attached in **Chyba! Nenašiel sa žiaden zdroj odkazov.**.

In terms of the requirements for the solution, by creating this part we have fulfilled the collection and visualization of SLA parameters from the clients' environments. In the context of the objective of the work, we have fulfilled the part concerning the appropriate interpretation of the overview of the status and load of the environment for the service user. An additional output is easing the work of administrators in terms of an overview of the projects and instances in the cloud.

6.7 Monitoring dashboard for the administrator

In this part, we will focus on creating a clear dashboard for the administrator. Its form and content were consulted with the administrators at KIS FRI UNIZA. Given the larger number of panels, we will present the dashboard step by step.

6.7.1 Visualization of the monitoring of *OpenStack* services

Regarding the *OpenStack* services, see also chapter 4.2, we are interested in visualizing their availability and health, since their outage leads to a deterioration in the quality of the service provided. Specifically, we will visualize the SLI for the availability of the individual *OpenStack* services in a time window defined by the administrator. This makes it possible to verify the SLI value for the required time windows, which can be useful in terms of retrospective checking of SLA compliance. Furthermore, we will display the health of the said services in words, so that the administrator can quickly identify whether they are functioning correctly. The last aspect of interest is the visualization of timelines on which the availability of services over time is plotted. This is a common way of presenting the availability of online services. From such a presentation, we can read exactly when a

service went down and how long its recovery took. For this purpose, we created several panels, which we describe below. The way they are added was covered in chapter 6.5.5.

We will first address the SLI panel for the availability of *OpenStack* services. For the purpose of calculating the SLI, binary metrics are available for each service. These indicate their availability. For demonstration, we show the query for determining the availability of the *nova* component: *avg_over_time(sum((openstack_nova_up or on() vector(0)))[[$__range:]) * 100*. The innermost expression is represented by the metric expressing the state of the service and a zero vector, which we need in the event of a physical outage of the node on which the service runs. In such an event, *openstack-exporter* is unable to obtain the metric value, which results in a *NODATA* entry in the given time window. For a correct calculation of availability, we need to fill these empty windows with the value zero, which is precisely what the zero vector is used for. With the *sum* function, we merge the zero vector with the metric vector, thereby obtaining an uninterrupted resulting vector. Over it, we calculate the average value in the time interval specified in the dashboard interface, which is represented by the variable `$__range`. To obtain a percentage value, we finally multiply the result by one hundred. The query settings for all panels are part of **Chyba! Nenašiel sa žiaden zdroj odkazov.**, file Panel overview.

The panels presenting the health of services work on the basis of the `openstack_<SERVICE>_agent_state` metrics, which return a list of the components of the given service and their status presented in binary form. We have already discussed these components in chapter 4.2. In the environment of the monitored platform, we can extract such information from the *nova, cinder* and *neutron* services. If all components are working correctly, we map this value to the word *HEALTHY*; if some of them are not working, we present this with the word *PROBLEM*; and if none of the components are working, we use the word *DOWN*. We can notice that **Figure 6.18** shows a problem for the *cinder* component. On these panels, we created links to panels of another dashboard, in which we display the details of the components, as indicated by the small windows in the upper left corners. The way to create such links is described in the documentation [66], specifically in *Dashboards->Build dashboards->Manage dashboard links*. The details of the *cinder* service are presented in **Figure 6.17**, where we can notice the non-functioning *cinder-volume*

Cinder

Time	adminState	hostname	job	service	zone	Status
2023-04-16 00:0...	enabled	block1@lvm	openstack_expor...	cinder-volume	nova	0
2023-04-16 00:0...	enabled	controller	openstack_expor...	cinder-scheduler	nova	1

Figure 6.17 Details of the cinder service

component. This also corresponds to the information from the administrators. It follows that this service is available, but not functional.

For the last panel, we have a *state timeline* chart available, to which we added the binary metrics presenting the availability of services, and *Grafana* did the rest automatically.

Figure 6.18 shows the resulting combination of all the panels mentioned. With this, we have completed the visualization of service availability and continue with other



Figure 6.18 Visualization of the monitoring of OpenStack services

components of the platform.

6.7.2 Visualization of reserved and available resources for the cloud

So that administrators have an overview of the total amount of resources provided in the cloud, we are interested in presenting this information visually. Since we are dealing with a cloud that primarily provides IaaS, we include vCPU, RAM and storage among the resources provided. As we discussed in 6.6, we provide clients with an overview of the resources used in their projects. From the administrator's point of view, it is important to see the total sum of resources used by all projects. On this basis, they can assess whether their platform has sufficient capacity for further customers. For clarification, we add that the visualized data do not present the current but the total resources in use. This means that if a client created a new VM with one vCPU, the total value of used vCPUs would increase by one. If the client shut down this VM, the value would not change, since the vCPU is still mapped as used. The total sum would decrease only when this VM is deleted.

We visualize these elements by means of three panels showing the total number of used vCPUs and the percentage utilization of memory and storage, see **Figure 6.19**. All information for these panels comes from the *nova* service. For the number of used vCPUs, we have a metric directly expressing this value. In the case of memory, we have metrics expressing the amount of used and available bytes. The *PromQL* query for obtaining the

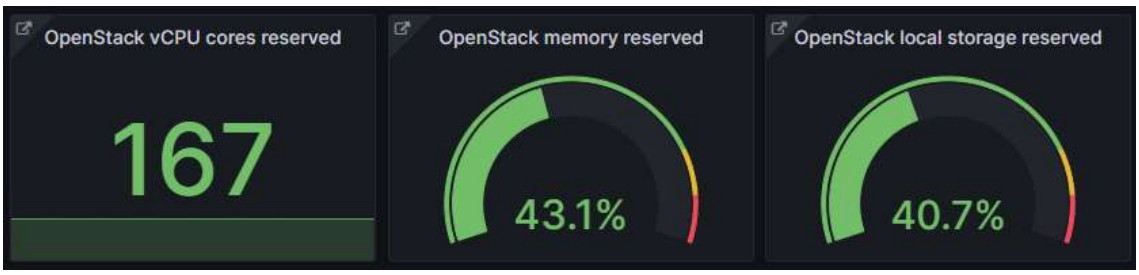


Figure 6.19 Resources used in the cloud

percentage utilization value looks as follows: `(openstack_nova_memory_used_bytes/(openstack_nova_memory_used_bytes+openstack_nova_memory_available_bytes)) * 100`. In the case of storage, the query is identical, only with the corresponding metrics.

6.7.3 Visualization of the monitoring of the physical infrastructure

We now move on to the visualization of data from the monitoring of the physical infrastructure, which is also one of the main objectives of this work. *Grafana* is very well designed for visualizing dynamic environments, which we also tried to take advantage of when creating the panels for the infrastructure. The benefit of such a design is the automatic visualization of data from newly added devices.

The visualization of the monitoring of the physical infrastructure forms the majority of the dashboard for the administrator. Owing to the extent and number of panels created, and at the same time in an effort to keep this work clear, we will not give examples of all the panels created. However, we make them available to the reader in the appendices, see **Chyba! Nenašiel sa žiaden zdroj odkazov.**, file Panel overview.

What exactly are we interested in visualizing in terms of the physical infrastructure? As we discussed in chapter 3.2, we need to display the utilization of CPU, RAM and storage on the physical nodes and the monitoring server. The administrators also asked us for an overview of all compute nodes and network devices in the infrastructure, where by overview we mean the number of functional and non-functional devices together with a list of them.

We provide administrators with two dynamic tables showing all compute and controller nodes within the cloud. In such a table, we present information about the node's IP address, current CPU utilization, average CPU utilization over the last fifteen minutes, and RAM used. In addition to the tables, we provide panels with information on storage utilization. If a standardized naming system for file systems existed, it would be possible to include storage in the dynamic tables. At present, however, it is necessary to select the relevant file systems manually for each node and set up their visualization. Let us also

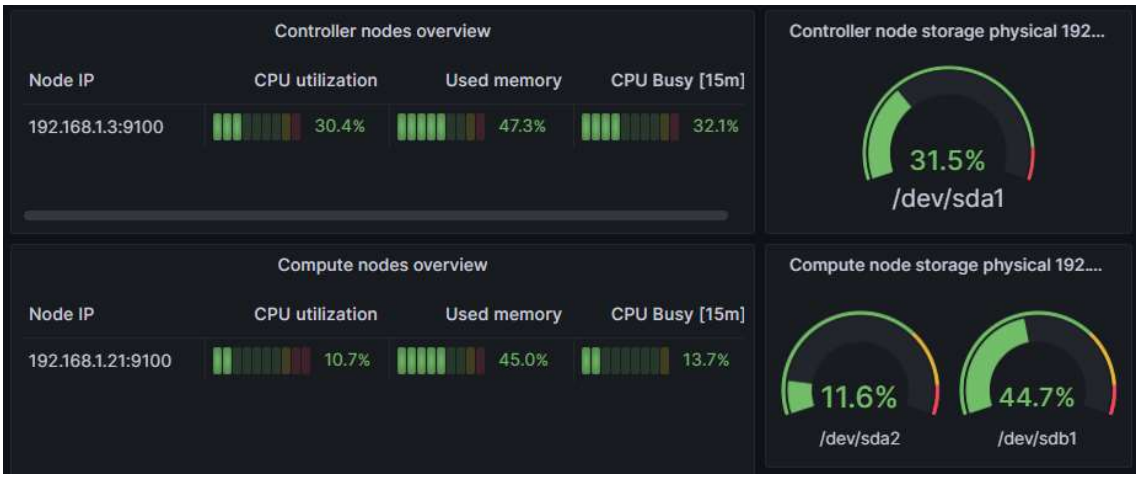


Figure 6.20 Visualization of resources used on the nodes

describe the dynamic nature of the said tables, which consists in the fact that after a new device is added to the corresponding job in the *Prometheus* configuration, see 6.4.8, the system automatically registers it and adds it to the table. An example of the visualization of used resources is presented in **Figure 6.20**.

For an overview of all devices in the infrastructure, we created clear panels presenting the number of functional and non-functional devices, separating compute nodes from

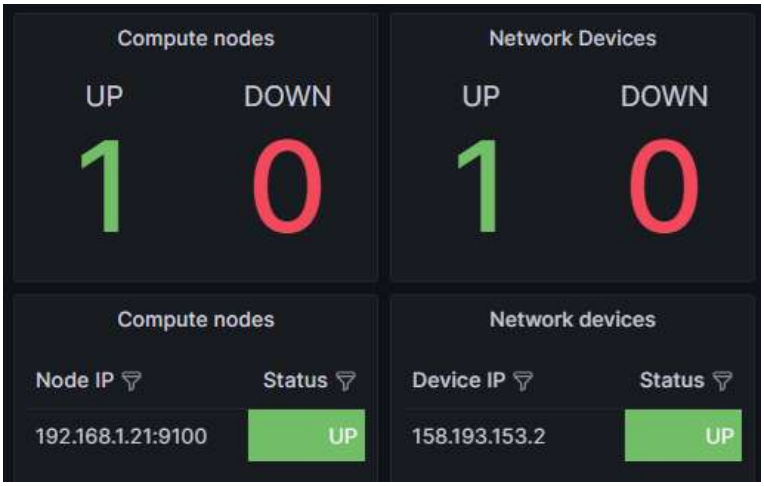


Figure 6.21 Overview of devices in the infrastructure

network devices. Based on the requirements, we also created a list of them. With these elements, we likewise kept their dynamic nature in mind, so they work on the same principle as we described for the tables above. For a better idea, **Figure 6.21** presents an example of the visualization created.

In addition to the main monitoring dashboard, we created two additional ones. A link to them is placed in the upper right corner of the main monitoring dashboard. These are *Nodes info*, providing a detailed, visualized overview of all metrics collected by the *Node exporter* exporter for the selected node. The *Network devices* dashboard provides the same functionality, but for network devices, i.e. for metrics collected by means of *snmp-exporter*. For these dashboards, we drew inspiration from downloaded templates, and they provide the administrator with a comprehensive overview of the operation of the devices, which can be useful when determining the origin of problems.

6.7.4 Other elements of the monitoring dashboard

In addition to the components discussed so far, we added two extra panels to the dashboard for administrators. One is an overview of all activated alerts within the *Old-OpenStack* folder. Thanks to this, the administrator can clearly see in which parts of the cloud there are current problems. We will discuss the topic of alerting further in chapter 6.8. We created the second panel at the request of the administrators, and it provides an overview of vCPU utilization by all instances in the cloud. The purpose of this panel is to provide a way of identifying an instance that exhibits high vCPU utilization over a longer period, which may foreshadow undesirable behaviour.

By creating a clear dashboard, we enable administrators to have a simple overview of the state of the cloud as a whole, and, based on the colour interpretation of the values, the administrator can assess whether the services provided are at risk. Formally, we could consider this as fulfilling the objective of the work relating to this topic; however, we consider it important that a process of active alerting is set up in the event of problems arising, which we address in the next part.

6.8 Alerting

Grafana provides a sophisticated method of alerting. The basic elements of this system are alert rules, contact points and notification policies. Below we briefly present our implementation of these elements.

6.8.1 Alert rules

In general, this part defines the conditions under which an alert is generated. Within our deployment, we created twenty-six rules, which we defined in seven evaluation groups. These group the rules either on the basis of their mutual relationship or a common evaluation interval. The list of rules created is given in **Table 6.3**, and to understand it correctly we need to define the abbreviations for compute node (VU), network devices (SZ), controller node (RU) and monitoring server (MS). For the network interface utilization rule, we create alerts only for the listed interfaces, since, based on information from the administrators, only these are included in the physical infrastructure of the cloud.

We determined the intervals as well as the conditions for evaluating the rules on the basis of experience gained in practice, since in researching the topic we did not come across any form of standardization.

Vyhodnovacia skupina	Interval vyhodnocovania	Pravidlá	Vyhodnotenie na základe
Dostupnosť	20 sekúnd	Dostupnosť VU	Posledná hodnota
		Dostupnosť SZ	
		Dostupnosť RU	
Vyťaženie hardvérových zdrojov	30 sekúnd	Vyťaženie CPU na VU	Priemer za posledných 5 minút viac ako 80% využitia
		Vyťaženie RAM na VU	
		Vyťaženie CPU na RU	
		Vyťaženie RAM na RU	
		Vyťaženie CPU na SZ	
		Vyťaženie RAM na SZ	
		Vyťaženie CPU na MS	
		Vyťaženie RAM na MS	
Využitie sieťových rozhraní	1 minúta	Využitie rozhraní Gi1/0/25-1/0/30	Priemer za posledných 5 minút viac ako 80% maximálnej priepustnosti rozhrania
Rezervované zdroje v OpenStack	10 minút	Rezervovaná RAM	Priemer za poslednú hodinu viac ako 80% z maxima
		Rezervované úložisko	
OpenStack služby	1 minúta	Dostupnosť služby Cinder	Posledná hodnota
		Dostupnosť služby Nova	
		Dostupnosť služby Heat	
		Dostupnosť služby Keystone	
		Dostupnosť služby Neutron	
		Dostupnosť služby Glance	
		Stav komponentov Cinder	
		Stav komponentov Nova	
		Stav komponentov Neutron	
Kontrola úložiska	30 minút	Využitie úložiska na VU	Priemer za poslednú hodinu viac ako 90% z maxima
		Využitie úložiska na RU	
Virtuálne inštancie	5 minút	Vyťaženie CPU na inštanciách	Priemer za posledenú hodinu viac ako 80% využitia

Table 6.3 List of rules

6.8.2 Contact points

Contact points define the target destination and the method of sending alerts, with a wide range of solutions supported, from simple e-mail, through corporate communication platforms, to *chat* applications.

In our case, we configured only one contact point, which we named *Main alerts*. Within it, we defined two targets, *Teams* for the administrators and *Discord* for testing. For both platforms, we used *webhook* technology, which is an HTTP-based function enabling communication between two API endpoints. In practice, it works in such a way that, when an alert is generated, *Grafana* sends a POST message about the occurrence of the event to the provided URL. This request is then processed by the end platform, and the user

receives a message with the alert. For a more detailed description of how this technology works in general, we recommend the reader the explanation from *RedHat*, see [72].

The process of creating a contact point for the platforms we used is very simple. It is necessary to generate a *webhook* URL for the desired channel, then in *Grafana* edit or create a new contact point, select the platform and insert the URL. In the *Teams* platform, such a URL can be obtained by navigating to the channel's additional options, where this option is directly located. However, it is not available for all types of channels, e.g. educational channels for students.

6.8.3 Notification policies

This concept is useful mainly in the case of large deployments, where alerts are expected to be sent to different administrator teams based on their content. The division can be performed by defining labels, which can be added in the settings of the individual alert rules.

This part is also where alert rules and contact points are linked. By default, a *root policy* is available, to which all alerts will be routed. In terms of our deployment, it is not necessary to create any further policies, since the scope of the deployment does not require it. We therefore added the contact point we created to the said default policy.

6.9 Testing

After the successful deployment of the components, creation of the dashboards and setting up of alerts, it is necessary to test the functionality of the system. Since the monitored

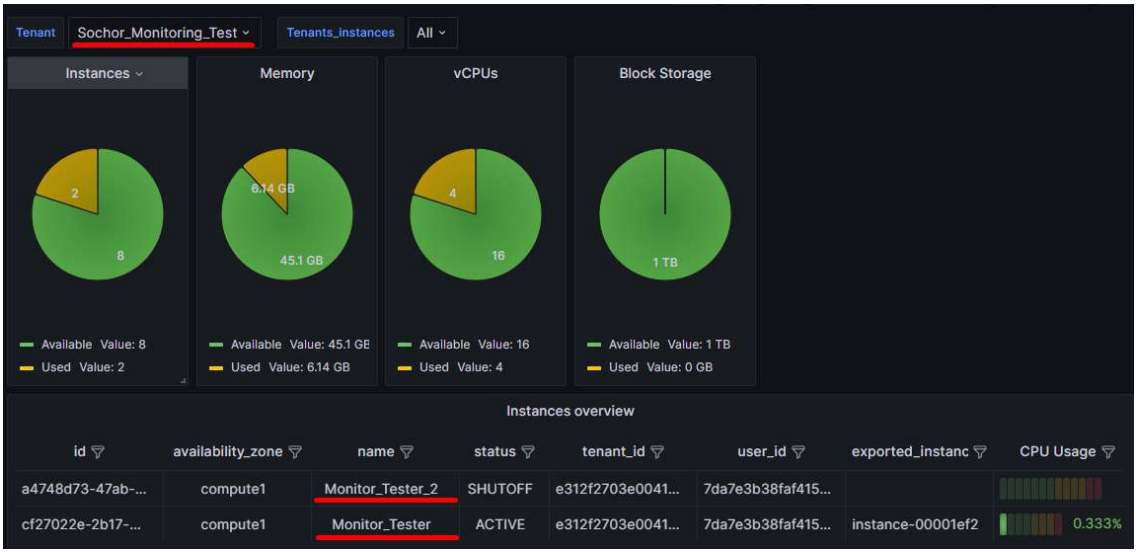


Figure 6.22 Verification of the *Sochor_Monitoring_Test* project

platform is in active use, we must be careful when testing it. Within our tests, we will verify the visualization and sending of alerts for the availability of both physical nodes and *OpenStack* services, resource load on the servers, and suspicious behaviour on VMs inside the cloud. For these purposes, we created in the monitored platform a project named *Sochor_Monitoring_Test* and two instances, *Monitor_Tester,* and *Monitor_Tester_2,* on which we also installed *node exporter*. This is also confirmed by data from the monitoring platform, since in the *Tenants overview dashboard* we can select this project in the *Tenant* variable, see **Figure 6.22**. In the figure we can also see the two instances created, one of which indicates the SHUTOFF state, which corresponds to reality, since we shut down this VM before taking the screenshot.

6.9.1 Resource load and availability

The most critical event within the infrastructure can be considered to be the outage or overload of a device. That is precisely why it is appropriate to verify whether the alert that has been set up works and how long it takes to reach the administrator. We will also verify whether the panel we set up will visualize this outage. We have already discussed dynamic panels in chapter 6.7.3; the rules for all devices work in the same way. To the configuration file of the job intended for compute nodes, we added the IP address of the *Monitor_Tester* instance, which will simulate a compute node for us for these purposes. As soon as *Prometheus* loads the configuration, the panels relating to compute nodes change on the monitoring dashboard, see **Figure 6.23**. A new entry was added alongside the original node and the number of nodes in the UP state increased.

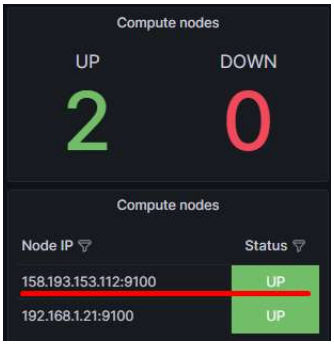


Figure 6.23 Adding a compute node

First, we will simulate CPU and RAM load. For this purpose, we will use the *stress-ng* tool, which allows us to load both components at once. The *Monitor_Tester* VM has two GB of RAM and two vCPU cores available. For the load, we used the command `#stress-ng --vm-bytes 1500M --vm-keep -m 2 --Timeou 420`, which created two VMs in the system to load the vCPU cores. In addition, one thousand five hundred MB of memory was allocated in RAM. More information on how the tool works and on its arguments is available in the program's manual. We started the testing at 11:00 local time. The load on the system is demonstrated by information from the *htop* tool, see **Figure 6.24**, but also by data directly from the monitoring dashboard, see **Figure 6.25**.

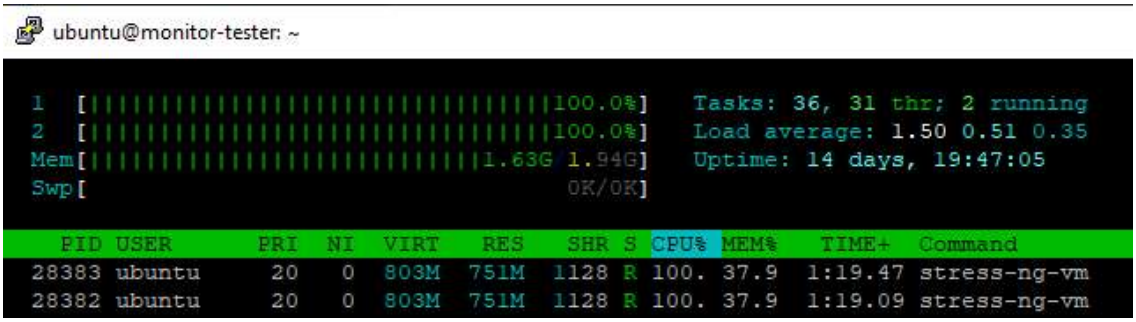


Figure 6.24 System load – *htop*

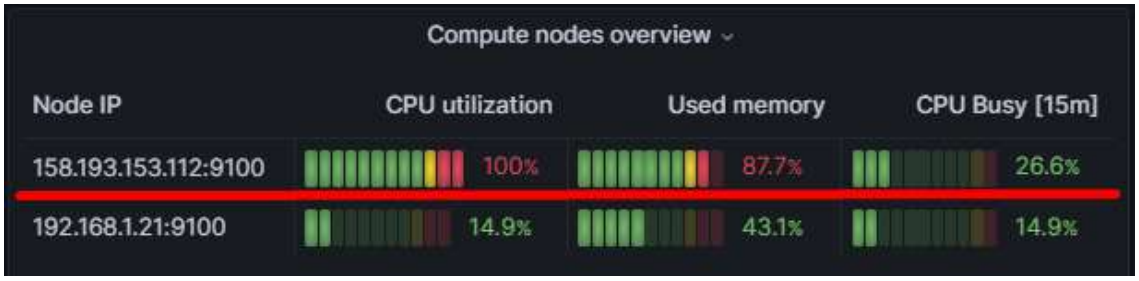


Figure 6.25 System load – Grafana

Based on the rule set for resource load, the average utilization over the last five minutes must rise above eighty percent. In our testing, the alerts were created at 11:05:30 for the CPU and at 11:06:00 for RAM, which can also be seen on the monitoring dashboard, panel for active alerts, see **Figure 6.26**.

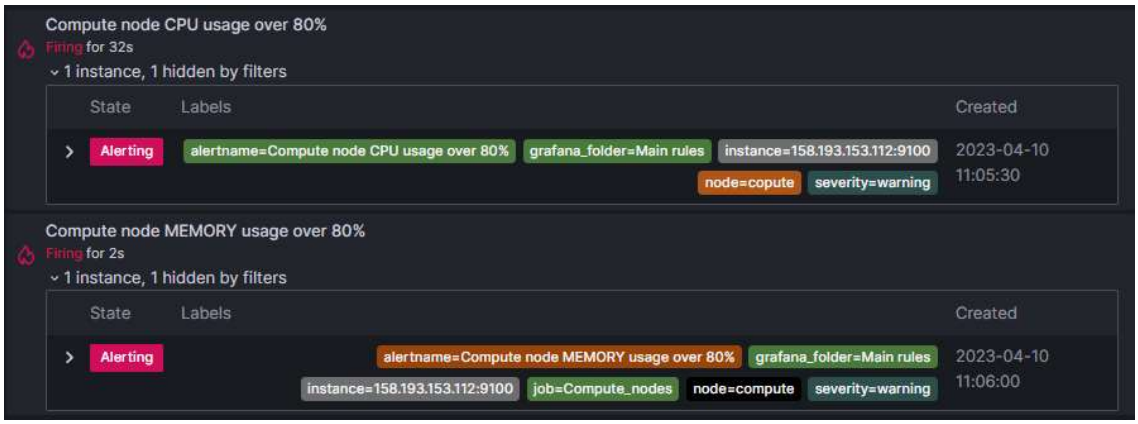


Figure 6.26 Created alerts – panel for active alerts

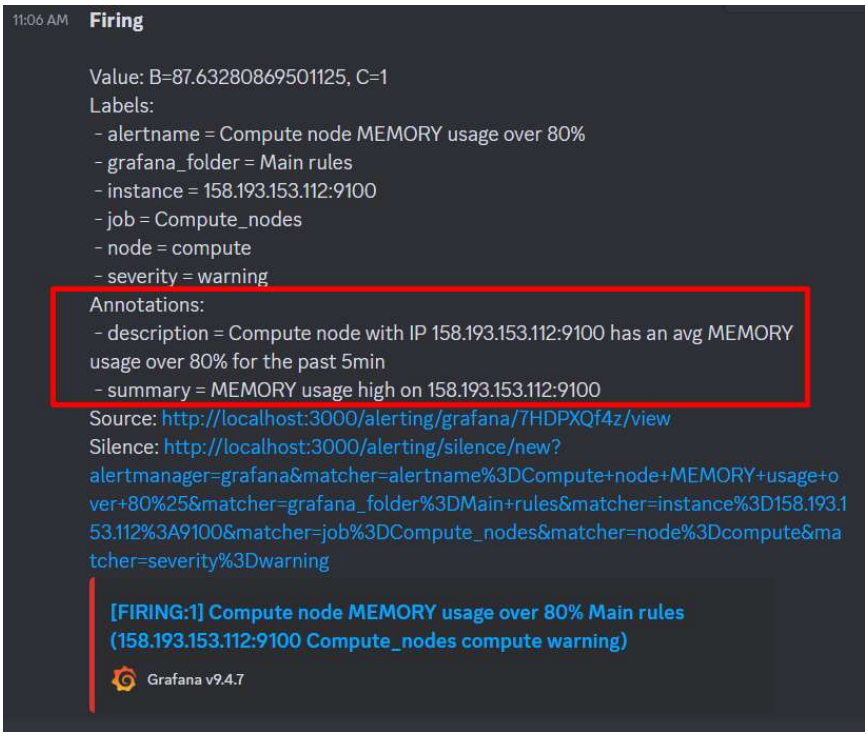


Figure 6.27 Generated alert – discord

The administrators were also notified of the event by text alerts being sent to the configured platforms. In the case of the *Discord* and *Teams* platforms, we received the event reports between 11:06 and 11:07. The text alert for high memory utilization is shown in **Figure 6.27**, where the highlighted part contains the most important information, such as

what happened and on which device, plus which component it concerns. After the load ended, when the average utilization dropped below eighty percent, the alerts were automatically resolved and the administrators were notified by a *Resolved* message. For cases where an alert is not dealt with for a longer period, we set the interval for repeated sending of text messages to four weeks. In practice, this means that if an alert remained active for a longer period, the message is sent when the alert is generated and then every fourth week. This interval can be changed in the notification policy settings, section *Timing options->Repeat interval*.

We will simulate the testing of the availability alert by shutting down a VM, while observing how long it takes for the alert to be generated. We shut down the VM at 11:50 local time, and within fifteen seconds the panels on the dashboard indicate the unavailability of the node, see **Figure 6.28**.

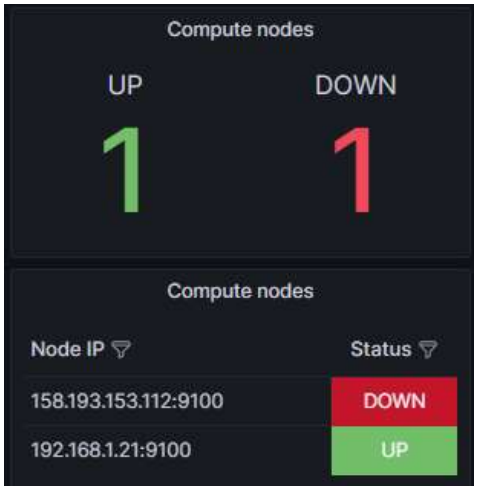


Figure 6.28 Node unavailability – monitoring dashboard

The alert was generated within twenty seconds of the VM being shut down, see **Figure 6.29**, and we received the text message within one minute of the outage.

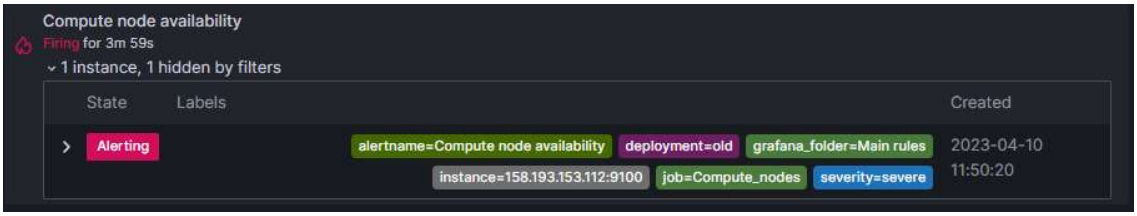


Figure 6.29 Alert – node outage

After the VM was restarted, the alert was resolved within twenty seconds and the dashboard again indicated the functional state of both nodes. We received the text message about the resolution of the problem within ten minutes of the recovery.

6.9.2 OpenStack services

Another component that we are interested in testing is alerting on the outage of *OpenStack* services. In the monitored cloud, this method of testing is difficult, since shutting down a service affects other users. In our case, however, we can use the *cinder* service, which is not fully functional in this deployment. We will carry out the testing by shutting it down and document the process of visualization and alerting.

Within the *cinder* service, we shut down its API component using `#service openstack-cinder-api stop`, thereby preventing any communication with this component. Within one minute of the shutdown, the dashboard signals the problem with the word *DOWN*. The value of service availability for the selected time window, in the case of **Figure 6.30** the last three hours, also decreases. At the same time, an alert is generated, see **Figure 6.31**, which we receive in text form within two minutes of the start of the outage.



Figure 6.30 Outage of the cinder service

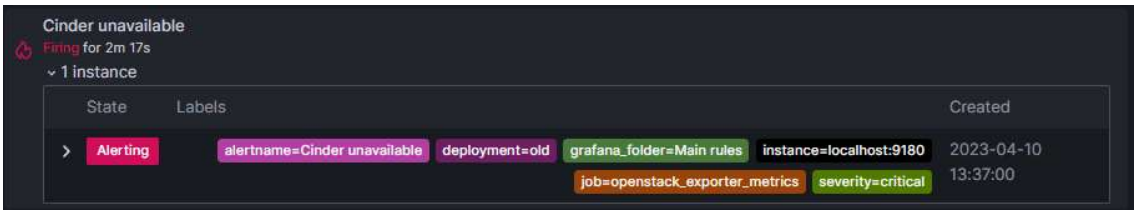


Figure 6.31 Alert for the outage of the cinder service

6.9.3 Suspicious VM behaviour

The administrators requested that a process be created so that they are informed in the event of long-term, high resource usage by instances in the cloud, since such a state

may indicate misuse of the provided resources for undesirable purposes. For the purposes of this testing, we will load the CPU on the *Monitor_Tester_2* VM and document the subsequent reactions. For the load, we use the command `#stress-ng --vm-keep -m 2`, thereby loading both vCPU cores. We start the test at 13:00 local time.

Thirty-five minutes after the start of the test, a pending alert was generated, see **Figure 6.32**, indicator *Pending*. This arises once the defined rule is met and transitions to an active alert after the configured pending period, which for this rule is one hour.



Figure 6.32 Pending alert for instance load

If the issue is not remedied during the configured hour, an active alert is generated



Figure 6.33 Active alert for instance load

and sent to the configured platforms.

CONCLUSION

The aim of the document was to design and implement a monitoring system for a cloud built on the *OpenStack* platform. On the theoretical level of addressing the topic, we first examined various motivations for creating a monitoring system and divided the perspectives on monitoring. One of the motivations included the verification of parameters defined in the SLA, so we addressed this topic and defined significant metrics within it. Subsequently, we identified important performance metrics of physical and virtual devices that could have an impact on the services provided, and we explained the way they are obtained. Since we were interested in monitoring the *OpenStack* platform, it was necessary to become familiar with it and introduce the components important for future monitoring. Towards the end of the theoretical part, we dealt with suitable ways of presenting the data obtained and introduced software tools suitable for the solution.

On the practical level, we established the requirements for the solution. On their basis, we chose the *Prometheus* toolkit and the *Grafana* visualization platform for the deployment of the system. We continued with the design of the architecture, consisting of a central server for data collection and exporters collecting metrics from API endpoints and devices. Based on the established architecture, we deployed the system. The process consisted of the installation and configuration of the individual components, the creation of visualization dashboards, and the setting up of alert rules that would be able to alert administrators when problems arise. Among the main monitored parameters, we included the utilization of CPU, RAM and storage on the physical nodes, together with their availability. We also monitor the availability of *OpenStack* services and resource usage by the individual instances in the cloud, for the purpose of their subsequent presentation to the users of the platform as well. In the final part, we tested the functionality of the system we deployed, thereby fulfilling the objective of this work. We also briefly addressed the possible future direction of the solution we created.

LIST OF REFERENCES

[1] "What is cloud computing?", 2012. https://aws.amazon.com/what-is-cloud-computing/ (cited Jan. 20, 2023).

[2] C. B. Hauser and S. Wesner, "Reviewing Cloud Monitoring: Towards Cloud Resource Profiling", *IEEE Int. Conf. Cloud Comput. CLOUD*, vol. 2018-July, pp. 678–685, 2018, doi: 10.1109/CLOUD.2018.00093.

[3] 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.

[4] L. Romano, D. De Mari, Z. Jerzak, and C. Fetzer, "A novel approach to QoS monitoring in the cloud", *Proc. - 1st Int. Conf. Data Compression, Commun. Process. CCP 2011*, pp. 45–51, 2011, doi: 10.1109/CCP.2011.49.

[5] 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.

[6] E. Aljoumah, F. Al-Mousawi, I. Ahmad, M. Al-Shammri, and Z. Al-Jady, "SLA in cloud computing architectures: A comprehensive study", *Int. J. Grid Distrib. Comput.*, vol. 8, no. 5, pp. 7–32, 2015, doi: 10.14257/ijgdc.2015.8.5.02.

[7] M. Singh, S. Bhushan, and S. Rani, "Investigation of SLA Management in Cloud Computing and Future Directions", *Proc. - 2021 2nd Int. Conf. Comput. Methods Sci. Technol. ICCMST 2021*, pp. 78–83, 2021, doi: 10.1109/ICCMST54943.2021.00027.

[8] S. Frey, C. Lüthje, R. Teckelmann, and C. Reich, "Adaptable Service Level Objective Agreement (A-SLO-A) for Cloud services", *CLOSER 2013 - Proc. 3rd Int. Conf. Cloud Comput. Serv. Sci.*, pp. 457–462, 2013, doi: 10.5220/0004356604570462.

[9] M. Alhamad, T. Dillon, and E. Chang, "Conceptual SLA framework for cloud computing", *4th IEEE Int. Conf. Digit. Ecosyst. Technol. - Conf. Proc. IEEE-DEST 2010, DEST 2010*, pp. 606–610, 2010, doi: 10.1109/DEST.2010.5610586.

[10] N. R. M. (O'Reilly) Betsy Beyer, Chris Jones, Jennifer Petoff, *Site Reliability Engineering*. Google, Inc., 2016.

[11] "AWS EC2 Cloud Watch documentation". https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/viewing_metrics_with_cloudwatch.html (cited Feb. 16, 2023).

[12] "AWS EC2". https://aws.amazon.com/ec2/ (cited Feb. 14, 2023).

[13] H. Lee, M. Kim, J. Hong, and G. Lee, "QoS parameters to network performance metrics mapping for SLA monitoring", *KNOM Rev.*, no. March, pp. 42–53, 2002, [Online]. Available at: http://mail.apnoms.org/knom/knom-review/v5n2/4.pdf.

[14] S. Lee, K. Levanti, and H. S. Kim, "Network monitoring: Present and future", *Comput. Networks*, vol. 65, pp. 84–98, 2014, doi: 10.1016/j.comnet.2014.03.007.

[15] J. Ellingwood, "Gathering Metrics from Your Infrastructure and Applications", 2017. https://www.digitalocean.com/community/tutorials/gathering-metrics-from-your-infrastructure-and-applications (cited Feb. 27, 2023).

[16] Brendan Gregg, "The USE Method". https://www.brendangregg.com/usemethod.html (cited Mar. 06, 2023).

[17] V. Lam, "3 Key Server Monitoring Metrics To Track For System Health And

Performance". https://theqalead.com/test-management/server-monitoring-metrics/ (cited Mar. 06, 2023).

[18] "Libvirt Manual Pages". https://www.libvirt.org/manpages/ (cited Mar. 18, 2023).

[19] "proc(5) — Linux manual page". https://man7.org/linux/man-pages/man5/proc.5.html (cited Mar. 18, 2023).

[20] B. Brazil, "Understanding Machine CPU usage". https://www.robustperception.io/understanding-machine-cpu-usage/ (cited Mar. 18, 2023).

[21] "Manual virsh". https://www.libvirt.org/manpages/virsh.html (cited Mar. 18, 2023).

[22] "How To Calculate CPU Utilization". https://www.embedded.com/how-to-calculate-cpu-utilization/ (cited Mar. 19, 2023).

[23] "Working With Memory Metrics From Node Exporter". https://acloudxpert.com/working-with-memory-metrics-from-node-exporter/ (cited Mar. 20, 2023).

[24] "GitHub Server Recommended alert thresholds". https://docs.github.com/en/enterprise-server@3.7/admin/enterprise-management/monitoring-your-appliance/recommended-alert-thresholds (cited Mar. 24, 2023).

[25] "Tableau Hardware Monitoring". https://help.tableau.com/current/blueprint/en-us/bp_hardware_monitoring.htm (cited Mar. 24, 2023).

[26] D. N, "Server Monitoring Best Practices". https://www.poweradmin.com/blog/monitoring-best-practices-2/ (cited Mar. 24, 2023).

[27] "Linux manual". https://man7.org/linux/man-pages/ (cited Mar. 23, 2023).

[28] "Check your disk space use with the Linux df command". https://www.redhat.com/sysadmin/Linux-df-command (cited Mar. 20, 2023).

[29] "Filesystem metrics from the node exporter". https://www.robustperception.io/filesystem-metrics-from-the-node-exporter/ (cited Mar. 23, 2023).

[30] European Telecommunications Standards Institute, "ETSI EG 202 765-3 V1.1.1 Speech and multimedia Transmission Quality (STQ); QoS and network performance metrics and measurement methods; Part 3: Network performance metrics", vol. 1, pp. 1–33, 2009.

[31] G. Gardikis, K. Sarsembagieva, G. Xilouris, and A. Kourtis, "An SNMP agent for active in-network measurements", *Int. Congr. Ultra Mod. Telecommun. Control Syst. Work.*, pp. 302–307, 2012, doi: 10.1109/ICUMT.2012.6459684.

[32] A. Ciuffoletti, "Monitoring a virtual network infrastructure: An IaaS perspective", *Comput. Commun. Rev.*, vol. 40, no. 5, pp. 47–52, 2010, doi: 10.1145/1880153.1880161.

[33] "What is OpenStack?" https://www.openstack.org/software/ (cited Feb. 04, 2023).

[34] 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.

[35] "OpenStack nova documentation". https://www.openstack.org/software/ (cited Feb. 04, 2023).

[36] “OpenStack swift documentation”. https://docs.openstack.org/swift/latest/ (cited Feb. 04, 2023).

[37] “OpenStack Monitoring”. https://docs.openstack.org/operations-guide/ops-monitoring.html (cited Mar. 16, 2023).

[38] “Nova System Architecture”. https://docs.openstack.org/nova/queens/user/architecture.html (cited Mar. 16, 2023).

[39] “Cinder System Architecture”. https://docs.openstack.org/cinder/latest/contributor/architecture.html (cited Mar. 18, 2023).

[40] “Networking service overview”. https://docs.openstack.org/neutron/latest/install/common/get-started-networking.html (cited Mar. 18, 2023).

[41] “OpenStack telemetry”. https://wiki.openstack.org/wiki/Telemetry (cited Apr. 01, 2023).

[42] “OpenStack Monasca”. https://wiki.openstack.org/wiki/Monasca (cited Apr. 01, 2023).

[43] H. Aljahdali, A. Albatli, P. Garraghan, P. Townend, L. Lau, and J. Xu, “Multi-tenancy in cloud computing”, *Proc. - IEEE 8th Int. Symp. Serv. Oriented Syst. Eng. SOSE 2014*, pp. 344–351, 2014, doi: 10.1109/SOSE.2014.50.

[44] S. Kajeepeta, “Multi-tenancy in the cloud: Why it matters”. https://www.computerworld.com/article/2517005/multi-tenancy-in-the-cloud-why-it-matters.html (cited Mar. 09, 2023).

[45] 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.

[46] “CSP vs. tenant - Understanding shared responsibilities”. https://www.kuppingercole.com/blog/reinwarth/csp-vs-tenant-understanding-shared-responsibilities (cited Mar. 10, 2023).

[47] “Reporting & Monitoring Overview”. https://developers.google.com/maps/reporting-and-monitoring/overview (cited Mar. 09, 2023).

[48] “IBM Cloud Pak System W3500 2.3.2.0”. https://www.ibm.com/docs/en/cloud-pak-system-w3500/2.3.2.0 (cited Mar. 11, 2023).

[49] “What is data visualization?” https://www.ibm.com/topics/data-visualization (cited Mar. 11, 2023).

[50] “Cloud monitoring”. https://cloud.google.com/monitoring/docs (cited Mar. 11, 2023).

[51] “Cloud monitoring guide: Response”. https://learn.microsoft.com/en-us/azure/cloud-adoption-framework/manage/monitor/response#successful-alerting-strategy (cited Mar. 13, 2023).

[52] “Deploy Azure Monitor: Alerts and automated actions”. https://learn.microsoft.com/en-us/azure/azure-monitor/best-practices-alerts (cited Mar. 13, 2023).

[53] R. Grati, K. Boukadi, and H. Ben-Abdallah, “Overview of IaaS monitoring tools”, *Proc. IEEE/ACS Int. Conf. Comput. Syst. Appl. AICCSA*, vol. 2016-July, 2016, doi: 10.1109/AICCSA.2015.7507146.

[54] “What is Graphite Monitoring?”, 2022. https://www.metricfire.com/blog/what-is-graphite-monitoring/ (cited Mar. 25, 2023).

[55] H. J, “Graphite Beginner’s Guide”. https://www.netadmintools.com/graphite-beginners-guide/#wbounce-modal (cited Mar. 25, 2023).

[56] “collectd – The system statistics collection daemon”. https://collectd.org/ (cited Mar. 26, 2023).

[57] “Grafana”. https://grafana.com/ (cited Mar. 26, 2023).

[58] “Zabbix manual”. https://www.zabbix.com/documentation/6.0/en/manual (cited Mar. 27, 2023).

[59] D. Lambert, “Multi-tenant monitoring: how to achieve that using free Zabbix open-source monitoring software”. https://blog.zabbix.com/multi-tenant-monitoring-how-to-achieve-that-using-free-zabbix-open-source-monitoring-software/12024/ (cited Mar. 27, 2023).

[60] “Prometheus documentation”. https://prometheus.io/docs/introduction/overview/ (cited Mar. 27, 2023).

[61] B. Brazil, “How much RAM does Prometheus 2.x need for cardinality and ingestion?”, 2019. https://www.robustperception.io/how-much-ram-does-prometheus-2-x-need-for-cardinality-and-ingestion/ (cited Mar. 20, 2023).

[62] “Prometheus node exporter”. https://github.com/prometheus/node_exporter (cited Mar. 29, 2023).

[63] “Prometheus snmp_exporter”. https://github.com/prometheus/snmp_exporter (cited Mar. 30, 2023).

[64] “openstack-exporter”. https://github.com/openstack-exporter/openstack-exporter (cited Mar. 30, 2023).

[65] “Cannonical prometheus-openstack-exporter”. https://github.com/canonical/prometheus-openstack-exporter (cited Apr. 02, 2023).

[66] “Grafana documenation”. https://grafana.com/docs/grafana/latest/ (cited Mar. 30, 2023).

[67] “Global OID reference database”. https://oidref.com/ (cited Apr. 02, 2023).

[68] “circitor MIB files repository”. https://www.circitor.fr/Mibs/Mibs.php (cited Apr. 02, 2023).

[69] “Go documentation”. https://go.dev/doc/ (cited Apr. 02, 2023).

[70] “Manpage collectd.conf(5)”. https://collectd.org/documentation/manpages/collectd.conf.5.shtml# (cited Apr. 02, 2023).

[71] “Understanding Systemd Units and Unit Files”. https://www.digitalocean.com/community/tutorials/understanding-systemd-units-and-unit-files (cited Apr. 01, 2023).

[72] “What is a webhook?” https://www.redhat.com/en/topics/automation/what-is-a-webhook (cited Apr. 07, 2023).

[73] “Grafana mimir”. https://grafana.com/oss/mimir/ (cited Mar. 20, 2023).

ANNEXES

Annex A: List of MIB files for the snmp_exporter generator

```
sochor@prometheus:/opt/snmp_exporter_generator/generator/mibs$ ls
7.2.50.0.18765.RELEASE-B100-MIB.txt      PAN-LC-MIB.md5
7.2.50.0.18765.RELEASE-CERTS-MIB.txt     PAN-LC-MIB.my
7.2.50.0.18765.RELEASE-IPVS-MIB.txt      PAN-PRODUCT-MIB.md5
7.2.50.0.18765.RELEASE-ONE4NET-MIB.txt   PAN-PRODUCT-MIB.my
AIRESPACE-REF-MIB                        PAN-TRAPS.md5
AIRESPACE-WIRELESS-MIB                   PAN-TRAPS.my
apc-powernet-mib                         PDU-MIB.txt
ARISTA-ENTITY-SENSOR-MIB                 PICO-IPSEC-FLOW-MONITOR-MIB.txt
ARISTA-SMI-MIB                           PICO-SMI-ID-MIB.txt
ARISTA-SW-IP-FORWARDING-MIB              PICO-SMI-MIB.txt
CISCO-MEMORY-POOL-MIB.mib                PRINTER-MIB-V2.txt
CISCO-PROCESS-MIB.txt                    servertech-sentry3-mib
CISCO-QOS-PIB-MIB.mib                    servertech-sentry4-mib
CISCO-SMI.mib                            SNMP-FRAMEWORK-MIB
CISCO-TC.mib                             SNMP-FRAMEWORK-MIB.mib
CyberPower.MIB                           SNMPv2-CONF.mib
ENTITY-MIB                               SNMPv2-MIB
ENTITY-SENSOR-MIB                        SNMPv2-SMI
ENTITY-STATE-MIB                         SNMPv2-SMI.mib
ENTITY-STATE-TC-MIB                      SNMPv2-TC
HCNUM-TC                                 SNMPv2-TC.mib
HCNUM-TC.mib                             SYNOLOGY-DISK-MIB.txt
HOST-RESOURCES-MIB                       SYNOLOGY-EBOX-MIB.txt
IANA-CHARSET-MIB.txt                     SYNOLOGY-FLASHCACHE-MIB.txt
IANA-IFTYPE-MIB.txt                      SYNOLOGY-GPUINFO-MIB.txt
IANA-PRINTER-MIB.txt                     SYNOLOGY-ISCSILUN-MIB.txt
IF-MIB                                   SYNOLOGY-ISCSITarget-MIB.txt
INET-ADDRESS-MIB                         SYNOLOGY-NFS-MIB.txt
Infrapower-MIB.mib                       SYNOLOGY-PORT-MIB.txt
IPV6-TC                                  SYNOLOGY-RAID-MIB.txt
ISDN-MIB                                 SYNOLOGY-SERVICES-MIB.txt
KEEPALIVED-MIB                           SYNOLOGY-SHA-MIB.txt
LIEBERT_GP_PDU.MIB                       SYNOLOGY-SMART-MIB.txt
LIEBERT_GP_REG.MIB                       SYNOLOGY-SPACEIO-MIB.txt
MIKROTIK-MIB                             SYNOLOGY-STORAGEIO-MIB.txt
NET-SNMP-MIB                             SYNOLOGY-SYSTEM-MIB.txt
NET-SNMP-TC                              SYNOLOGY-UPS-MIB.txt
PAN-COMMON-MIB.md5                       UBNT-AirFiber-MIB
PAN-COMMON-MIB.my                        UBNT-AirMAX-MIB.txt
PAN-ENTITY-EXT-MIB.md5                   UBNT-UniFi-MIB
PAN-ENTITY-EXT-MIB.my                    UCD-SNMP-MIB
PAN-GLOBAL-REG-MIB.md5                   VRRP-MIB
PAN-GLOBAL-REG-MIB.my                    VRRPv3-MIB
PAN-GLOBAL-TC-MIB.md5                    WIENER-CRATE-MIB-5704.txt
PAN-GLOBAL-TC-MIB.my
```