



**FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE**

Assignment of master's thesis

Title: Monitoring of data-oriented applications in the hospital environment
Student: Bc. Vladimír Cherkezov
Supervisor: Ing. Tomáš Nováček
Study program: Informatics
Branch / specialization: Web Engineering
Department: Department of Software Engineering
Validity: until the end of summer semester 2022/2023



Instructions

The aim of this work is to design and implement a solution for monitoring data-oriented applications in the hospital environment. In this environment, it is not possible to use standard methods because the applications are available only in the internal hospital network.

The new solution should include the following functions:

- monitoring of the data processing in the hospital environment,
- monitoring of server resources (memory, disk space, etc.),
- sending the collected data to a collection server that is outside the hospital environment.

Follow these steps:

- 1) Analyze problems and constraints of the already implemented solution.
- 2) Analyze customer requirements for a new solution.
- 3) Analyze technologies, concepts, and components that will be used in a new solution.
- 4) Design architecture, data model, and API of services.
- 5) Implement a monitoring service that will run in a hospital environment.
- 6) Implement a collection service that will receive data from monitored environments.
- 7) Verify and test the correctness of design and implementation using appropriate procedures.
- 8) Fix the identified deficiencies.
- 9) Evaluate the result of the work and discuss the possibilities for further development.

Master's thesis

**MONITORING OF
DATA-ORIENTED
APPLICATIONS IN THE
HOSPITAL
ENVIRONMENT**

Bc. Vladimir Cherkezov

Faculty of Information Technology
Software Engineering Department
Supervisor: Ing. Tomáš Nováček
January 2, 2023

Czech Technical University in Prague

Faculty of Information Technology

© 2022 Bc. Vladimir Cherkezov. All rights reserved.

This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis: Cherkezov Vladimir. *Monitoring of data-oriented applications in the hospital environment*. Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2022.

Contents

Acknowledgments	viii
Declaration	ix
Abstrakt	x
Acronyms	xii
Acknowledgments	1
1 Analysis	3
1.1 Analysis of existing system	3
1.1.1 Basic terms	3
1.1.2 Analysis of components	3
1.1.3 Problems of existing solution	5
1.2 Requirements analysis	5
1.2.1 Actors	5
1.2.2 Functional requirements	6
1.2.3 Non-functional requirements	7
1.2.4 Requirements analysis summary	7
1.3 Docker monitoring	8
1.3.1 Docker introduction	8
1.3.2 Monitoring via stats command	8
1.3.3 Monitoring via pseudo-files	9
1.3.4 Monitoring via API	10
1.3.5 Docker monitoring summary	11
1.4 Bare metal monitoring	11
1.4.1 Monitoring via commands	11
1.4.2 Monitoring via pseudo-files	11
1.4.3 Bare metal monitoring summary	12
1.5 Monitoring systems	12
1.5.1 cAdvisor	13
1.5.2 Prometheus	13
1.5.3 TICK stack	16
1.5.4 Monitoring systems summary	17
1.6 Communication security requirements	18
1.6.1 Authentication	18
1.6.2 Privacy and Integrity	21
1.6.3 Security conclusion	22
1.7 Data and Metrics visualization	22
1.7.1 Grafana	22
1.7.2 TICK stack components	22
1.7.3 Prometheus	23
1.7.4 Visualization summary	23

1.8	System failure prediction	23
1.8.1	Thresholds method	23
1.8.2	Forecasting methods	24
1.8.3	System failure prediction summary	24
1.9	Data consistency	25
1.9.1	Insert synchronization	25
1.9.2	Update synchronization	26
1.9.3	Delete synchronization	26
1.9.4	Data consistency summary	27
1.10	Technology Analysis	27
1.10.1	Java & Kotlin comparison	27
1.10.2	Maven & Gradle comparison	28
1.11	Analysis summary	29
2	Design	31
2.1	Solution design	31
2.1.1	Prometheus solution	33
2.1.2	TICK stack solution	34
2.1.3	Solutions comparison	35
2.1.4	Solutions design summary	37
2.2	Domain model	37
2.3	Synchronization design	38
2.3.1	Naive synchronization solution	38
2.3.2	Remember state synchronization solution	39
2.3.3	Synchronization design summary	40
2.4	Design summary	40
3	Implementation	41
3.1	Monitoring solution infrastructure	41
3.1.1	Introduction	41
3.1.2	Monitoring server	43
3.1.3	Telegraf instance	44
3.1.4	Monitoring database	46
3.2	Collection solution infrastructure	46
3.2.1	Introduction	46
3.2.2	Collection server	48
3.2.3	Collection database	48
3.2.4	InfluxDB instance	48
3.2.5	Grafana instance	49
3.2.6	Conclusion	50
3.3	Synchronization	51
3.3.1	Data synchronization	51
3.3.2	Metrics synchronization	53
3.4	Security	53
3.4.1	Data security	53
3.4.2	Metrics security	55
3.5	Visualization	56
3.5.1	Dashboards	57
3.5.2	Panels	57
3.6	Failure prediction	59
3.6.1	Thresholds method	59
3.6.2	Holt-Winters seasonal method	60

3.7	Implementation summary	61
4	Testing	63
4.1	Integration tests	63
4.1.1	Monitoring server integration tests	63
4.1.2	Collection server integration tests	66
4.2	Tests coverage	67
4.3	Testing summary	68
5	Conclusion	71
A	Grafana dashboard examples	73
B	Grafana panel examples	75
	Contents of enclosed media	83

List of Figures

1.1	The existing solution	4
1.2	Prometheus architecture [11]	13
1.3	TICK stack architecture [17]	16
1.4	OAuth protocol flow [28]	20
1.5	Insert synchronization example	25
1.6	Update synchronization example	26
1.7	Delete synchronization example	26
2.1	The common architectures part	32
2.2	Prometheus solution architecture	33
2.3	TICK stack solution architecture	34
2.4	TICK stack solution modification	36
2.5	Domain model	37
2.6	Naive synchronization solution	39
2.7	Remember state synchronization solution	40
3.1	Thresholds method example	59
3.2	Holt-Winters seasonal method example	60
4.1	The collection server tests coverage	67
4.2	The monitoring server tests coverage	68
A.1	The hospital Docker monitoring dashboard	74
B.1	CPU usage panel creation example	76
B.2	ETL jobs received panel creation example	77
B.3	Holt-Winters based panel creation example	78

List of Tables

3.1	Visualization summary table	57
-----	---------------------------------------	----

List of code listings

1.1	docker stats command output example [1]	8
1.2	memory.stat pseudo-file content example [2]	9
1.3	Docker API request example [3]	10
1.4	free command calling example	11
1.5	Memory pseudo-file content example	12
1.6	Instant vector selectors example [4]	15
1.7	Range Vector Selectors example [4]	15
1.8	Float literals example [4]	15
3.1	Docker Compose file for monitoring solution	42
3.2	jib plugin usage	43
3.3	configuration.env for monitoring server	43
3.4	application.properties file with environment variables	44
3.5	The general Telegraf settings	44
3.6	The input plugins Telegraf settings	45
3.7	The output plugins Telegraf settings	46
3.8	Docker Compose file for collection solution	47
3.9	The InfluxDB instance configuration	49
3.10	The InfluxDB data source configuration for the Grafana	50
3.11	The synchronization state inplementation	51
3.12	The synchronization from monitoring server side	52
3.13	The synchronization from collection server side	52
3.14	Collection and monitoring servers security configuration	54
3.15	Collection server authorization	54
3.16	Collection server security configuration	55
3.17	The Telegraf side security configuration	56
3.18	The InfluxDB side security configuration	56
3.19	Grafana's panel creation InfluxQL example	57
3.20	Grafana's panel creation SQL example	58
3.21	Grafana's panel creation multiquery example	59
4.1	The mock of collection server	64
4.2	Basic integration test for synchronization process	64
4.3	The integration test for synchronization process failure	65
4.4	Basic integration test for batch creation process	66
4.5	The collection solution testing using the MockMvc	66

Acknowledgments

I want to thank all those who participated in the creation of this work. Without your help, this work would not exist.

I express my sincere gratitude to the supervisor of my thesis work, Ing. Tomáš Nováček, for valuable advice, suggestions, comments, and the time devoted to me while writing my thesis.

Next, I would like to express my deep gratitude to the Alpha company, in particular Ivana Sixtová, for providing an excellent topic and for advice and support in implementing the practical part of this work. Thank you for the opportunity to create something that makes sense.

I thank my parents, girlfriend, and friends for giving me help and support throughout the path, being a source of motivation for me, and getting along with my sometimes complex nature. Without you, this path would have been much more difficult.

I also thank Microsoft for creating the Microsoft Sculpt keyboard, thanks to which all my upper limbs retained their working capacity and were able to complete this diploma. Thank you, finally a product that doesn't cause pain.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. I further declare that I have concluded an agreement with the Czech Technical University in Prague, on the basis of which the Czech Technical University in Prague has waived its right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60(1) of the Act. This fact shall not affect the provisions of Article 47b of the Act No. 111/1998 Coll., the Higher Education Act, as amended.

In Praze on January 2, 2023

.....

Abstrakt

Tato diplomová práce je dedikovaná návrhu, tvorbě a testování monitorovacího řešení. Účelem této práce je vytvořit pro organizaci Alpha monitorovací řešení, které by umožnilo monitorovat servery nemocnic z prostředí nepřípojeného do jejich lokální sítě.

V rámci této práce my prostudujeme celý proces od vytěžení dat až po zjištění správnosti běhu serveru na jejich základě. To znamená, že budeme studovat oblasti jako synchronizace dat, bezpečnost, predikce selhání atd. Prozkoumáme také technologie, které nám tento proces zjednoduší. Během analýzy technologií pak vzniknou dvě nezávislé řešení: na základě Prometheus a na základě TICK stacku.

Řešení, které je výsledným produktem práce, využívá TICK stack a skládá se ze dvou oddělených komponent, které spolu komunikují. Jedná se o monitorovací část nemocničního serveru, která získává data z nemocničního prostředí, a o server na sběr dat, který sbírá data z monitorovacích částí a analyzuje je pro různé nemocnice. Budeme jej také nazývat monitoring a collection řešení.

Pro úplnější monitorování nemocničního prostředí existují dva typy dat zasílaných mezi těmito komponentami: aplikační data a metriky. Odesílání metrik je realizováno pomocí TICK stacku a odesílání aplikačních dat je implementované jako dvě aplikace na každé straně komunikace mezi monitoring a collection řešeními. Obě jsou napsané v Kotlinu s využitím frameworku Spring. Tyto komponenty jsou také podrobně testovány v závěrečné části naší práce.

Klíčová slova monitorování systémů, TICK stack, Spring framework, Kotlin, Prometheus, synchronizace dat, zabezpečení aplikace, predikce selhání

Abstract

This master's thesis is dedicated to monitoring solution design, creation, and testing. The purpose of this work is to create for the Alpha organization a monitoring solution that would allow observing the hospital servers from outside of their local network.

We will study the whole process, from extracting data to determining the correctness of the server operation based on them. It means that we will study areas such as data synchronization, security, failure prediction, etc. We will also explore technologies that simplify this process for us. The technologies are separated into two independent solutions: using the Prometheus and using the TICK stack.

The solution, which is the final product of the work, uses the TICK stack and consists of two separate components that communicate with each other. These are the hospital server monitoring part, which extracts data from the hospital environment, and the collection metrics server part, which collects data from the monitoring part and analyzes them for the different hospitals. We also will call them monitoring and collection solutions.

For more complete monitoring of the hospital environment, there are two types of data sent between these components: application data and metrics. The metrics sending is implemented using the TICK stack, and the application data sent is implemented as two applications on each communication side. Both of them are written in Kotlin using the Spring framework. These components are also tested in detail in the final part of our work.

Keywords system monitoring, TICK stack, Spring framework, Kotlin, Prometheus, data synchronization, communication security, failure prediction

Acronyms

AMPQ	Advanced Message Queuing Protocol
API	Application Programming Interface
ARIMA	Autoregressive Integrated Moving Average
AWS	Amazon Web Services
CPU	Central Processing Unit
DBMS	Database Management System
ETL	Extract, Transform, Load
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
I/O	Input/Output
JAR	Java ARchive
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
OOP	Object Oriented Programming
OS	Operating System
PHP	Hypertext Preprocessor
PKI	Public Key Infrastructure
PYPL	Popularity of Programming Language Index
RAM	Random-Access Memory
REST	Representational State Transfer
SAML	Security Assertion Markup Language
SQL	Structured Query Language
UI	User Interface
URL	Uniform Resource Locator

Introduction

Monitoring is an important and required part of the application life. As the application grows, its infrastructure becomes more complex. It becomes harder to search for the problems that arise during the application running. Typically, the result of the problems' appearance became a slowdown in the application, partial or complete application failure.

Searching for the problem becomes a non-trivial task that takes much time when the infrastructure is complex. In this case, it is very important to have all the necessary information to solve the problem quickly. However, to do this, it is necessary to implement fairly complete monitoring of everything that happens, from monitoring the application itself to monitoring the infrastructure and hardware on which the application is running.

Providing information about a problem is not the only benefit of monitoring. The very important benefit is also anomaly and failure prediction based on the received data. It allows you to solve your system problems before they occur.

Monitoring is an important topic for Alpha organization, which develops a system for hospitals, that automatically collects and analyzes patients' data. For this kind of application, detecting problems quickly or predicting them is extremely critical. Because the work of the application is directly related to the doctors' work, it is very important to minimize the number of application crashes.

The Alpha organization decided to create a solution for monitoring hospital environments. This solution should have not only monitoring requirements but also some important limitations associated with the domain itself. Among them, it is worth noting that the data should stay on the hospital server, and only the hospital server can realize communication. In other words, you cannot send requests to the hospital server. The reason is the typical infrastructure of a hospital network, where the target hospital server is located in the intranet. The other important restriction is that the solution involves monitoring several hospital servers. Together, these restrictions make the final solution non-trivial.

Purpose of work

This work aims to design and implement a solution that allows observing hospital servers' state from the outside. This work is focused on creating a solution that would fulfill all the restrictions associated with the domain's specifics and fit into the already existing infrastructure.

Among the smaller goals, we should highlight the analysis of an existing solution, the study of monitoring, and issues needed to meet the requirements for the future system. Next, we should design, configure and implement the components of a new solution and then test it.

Ultimately, this work should result in a working solution that fulfills all requirements and can be used to monitor hospitals' environments for the Alpha organization.

Structure of work

The work is structured into four logical parts: analysis, design, implementation, and testing.

The first part is focused on obtaining the information necessary to create a future solution. At the beginning of this chapter, we get acquainted with the already existing system, identify its problems and, based on this, analyze the requirements in relation to the future solution.

Next, we will study in detail the types of metrics that we need to track, figure out how to get them and what components to use to make collecting metrics easier.

Then, in accordance with the requirements analysis, we analyze more general areas, such as data synchronization, application security, failure prediction, and metrics visualization. A technology analysis was also carried out, which was necessary to determine the technologies used following the requirements analysis.

In the second part, we design future solutions and choose the most suitable version. We also design more specific features of the future solution, such as synchronization and the domain model.

The third part describes what has been implemented as part of our work. It includes configuring the components used to collect, store, visualize metrics, and perform failure prediction based on them. Further, the components involved in the collection and synchronization processes were also implemented. In addition to synchronization, problems were solved, such as communication security, authentication, authorization, data validation, and many others.

The fourth part is related to testing the components that were implemented. This section describes integration tests and test coverage of them.

Finally, we summarize the results of this work and suggest possible improvements.

Chapter 1

Analysis

This chapter will explore the information needed to create a solution for monitoring hospitals' servers. We will get acquainted with the existing solution and list requirements for a future solution. After that, we will find all information needed to meet these requirements.

1.1 Analysis of existing system

In this section, we will analyze the system that Alfa company already has, describe its components and denote some problems with this solution. The analysis of requirements will be created according to this section.

1.1.1 Basic terms

So before we start the analysis, let us explain the meaning of some used terms. It will help the reader avoid ambiguity when reading this work.

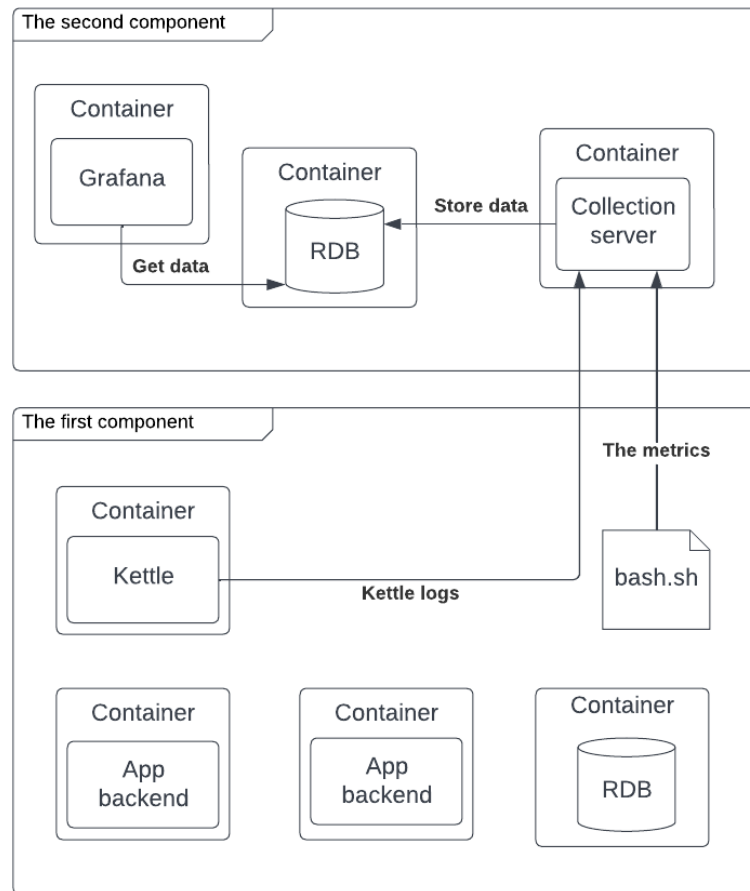
Collected data This term is related to the correct work of the system. In this case, we are talking about metrics, errors, and warnings of the system and not about personal data or data related to the outside world. The primary purpose for collecting this data is to monitor the system itself. When we mention data as part of the description of the collection solution or monitoring solution, it is also about collecting data.

Hospital environment By the concept of a hospital environment, we mean a host OS system of a hospital server and the related infrastructure, for example, a Docker engine.

Environment metrics By environment metrics, we mean the metrics provided by the hospital server and its infrastructure.

1.1.2 Analysis of components

First, we should familiarize the reader with the architecture of the existing solution. The existing solution consists of two separate components, that communicate with each other. You can see these components in the following figure:



■ **Figure 1.1** The existing solution

The first of them is within the hospital server and consists of several Docker containers. Among other Docker containers, we will highlight the backend of the application, the frontend of the application, the database, and the Pentaho Data Integrator instance, also known as Kettle.

The second component consists of three Docker containers: the Grafana instance, the database, and the collection server, which receives the collected data from the first component.

The collected data, for now, is the Kettle logs and some basic metrics from the host system. The Kettle logs are collected by the Kettle instance and sent by it to the collection server. There is a simple shell script for metrics collection that collects metrics from the OS and sends them to the collection service. These metrics are very basic and contain Disk, CPU, and memory metrics.

On the second component side, the collected data are stored in the database and, depending on them, are created visualizations and alerts in the Grafana instance.

In this work, the most important for us will be the first component, and we will focus on collecting information from its components. It is not important how these components run in our work, and that's why some operation details of these components can be omitted.

In the framework of this work, we will be interested in the following information about the components:

- What component data will be sent to the collection server
- Details about how component runs: using containers, bare metal, and others

The components we want to describe are the following:

Kettle This component runs as a separate Docker container in the hospital environment. It produces information about jobs, which it processes in several steps.

Application The Application consists of two parts: Application frontend and Application backend. These parts run separately in Docker containers. Our solution assumes that the Application will only send exceptions from the backend.

Application Database The Application database also runs as a separate Docker container and sends nothing. That means it is interesting for us only for hospital environment data collection.

Monitoring script This component runs bare metal and collects basic metrics from the hospital environment, this component. We will not work with the data of this component, and this component will appear only in the analysis part.

1.1.3 Problems of existing solution

In this subsection, we will describe the problems of the existing solution. The existing solution had several disadvantages:

- The data are sent only once a day.
- The application data are not sent.
- The data are not processed and a large amount of unnecessary information is sent over the network.
- The solution contains the collection of only basic metrics about the hospital environment.

These problems prompted us to create a more thoughtful and complex solution that would solve these problems. So it is the reason why was created the requirements analysis.

1.2 Requirements analysis

This section describes the processed functional and non-functional requirements of the Alfa organization. Depending on functional requirements, we will determine how and what actions the components of our solution should perform, while non-functional requirements will impose restrictions on the implementation of the future solution.

Based on these requirements, an analysis will be carried out, and a future solution will be proposed and implemented. The final solution will be based on these requirements.

1.2.1 Actors

For requirements analysis, we will define two types of actors. Actors are services, in our case, not users. These actors will be named *the monitoring solution* and *the collection solution*. Next, we will give them a detailed definition.

Monitoring solution

As part of the analysis, we will use the term monitoring solution to denote a group of components (or one component) whose main task will be to collect information within the hospital environment. The monitoring solution will also send information collected inside the hospital environment to the collection solution, which is outside of the hospital environment.

Collection solution

As part of the analysis, we will use the term collection solution to denote a group of components (or one component) whose main task will be to collect information that it will receive from the monitoring solutions. The collection solution can also process data, for example, visualize it, analyze it, and more. It is also correct to specify that the collection solution should receive information from several monitoring solutions from different hospital environments.

1.2.2 Functional requirements

In this subsection, we will describe functional requirements, which define individual actions and activities that must be performed.

F1 Authentication

- The monitoring solution will be able to connect to the collection solution using the requested credentials.

F2 Data collection

- The collection solution will be able to collect data, which the monitoring solution will send.
- The monitoring solution will be able to send data to the collection server periodically in batches.
- The monitoring solution will be able to collect data from the hospital environment.
- The monitoring solution will be able to collect hospital environment metrics that will be used for fault prediction and troubleshooting.

F3 Data storage

- The monitoring solution must store data that will be collected from the hospital environment.
- The collection solution must store data collected from the monitoring solution.

F4 Visualization

- The collection solution will be able to visualize data that will be collected from the monitoring solution.

F5 Failures prediction

- The collection solution will be able to predict failures and critical errors in the hospital environment.
- The collection solution will be able to send notifications about failures and critical errors in the hospital environment.

1.2.3 Non-functional requirements

This subsection will describe non-functional requirements. Here we will determine the constraints and requirements for our solution.

N1 Data consistency

- The collected data must be consistent between monitoring and collection solutions over time, meaning that data should not be lost or duplicated.

N2 Failure requirement

- After a failure of the monitoring solution or the collection solution, the data in databases must be synchronized correctly.

N3 Communication

- Communication must be secured.
- Communication must be one-directional. It means that the monitoring solution can request a collection solution but not vice versa.
- Communication should work for several monitoring solutions. It means that the collection solution should receive information from several monitoring solutions.

N4 Time zones requirement

- The Domain model should be designed in such a way as to allow monitoring solutions to be located in different time zones.

N5 Technologies

- Grafana should be used for data visualization.
- PostgreSQL should be used for storing the collected data.
- JVM 11 or higher should be used for monitoring and collection server implementation.
- Spring framework must be used for the monitoring and the collection servers implementation.

1.2.4 Requirements analysis summary

Depending on requirements analysis and analysis of the existing system, we need to define some issues we need to solve.

First, we need to know what environment metrics we can collect from our environment. A feature of the existing solution is Docker usage, and it will be important for us to find out what metrics it can provide about containers running in it.

However, we assume that the load on the system may occur by processes not associated with the Docker environment. So, for our solution, two issues deserve detailed analysis: Docker and bare metal monitoring.

It is also necessary to find out if there are already solutions that can help us collect metrics and consider integrating them before our solution.

Depending on requirements, we can identify other tasks that deserve research, such as communication security, data consistency between collection and monitoring solutions, and prediction of failure on the monitoring solution side. Analyzing these problems should give us an exhaustive amount of knowledge for successfully designing a new solution.

Finally, we should analyze the technologies we can choose for the monitoring and the collection servers implementation.

1.3 Docker monitoring

This chapter is dedicated to learning about metrics provided by Docker and ways how to get these metrics.

This section assumes that you have a basic knowledge of how Docker works. Without this knowledge, understanding the following material will be difficult. Let us provide a brief introduction to Docker technology.

1.3.1 Docker introduction

Docker is an open platform for developing, shipping, and running applications. Docker allows us to decouple our applications from our infrastructure through a loosely isolated environment called a container. The container, in turn, contains everything we need to run our application. Thus, Docker provides the ability to package and run an application regardless of our host configuration.

Let us take a look at how a Docker container is created. The way how to create a container is to use the image. An image is a read-only template with instructions for creating a Docker container. The image is usually based on some Linux-based operating system.

Images, containers, and other Docker objects are managed by the Docker daemon, which is a part of Docker architecture. Docker uses a client-server architecture, and the Docker daemon is a server part of this architecture, and on the client side is the Docker client. They can be called using the console as `docker` and `dockerd`.

Through the Docker API, the Docker client interacts with the Docker daemon. For example, when we run the command `docker run`, the Docker client requests the Docker daemon to run some image as a container.

It was a relatively brief introduction to Docker. For more details, please check out the official documentation. [5]

Now, let us look at how to get data about the Docker containers. Docker provides three ways to get metrics, and we propose to consider them in detail.

1.3.2 Monitoring via stats command

Docker provides a special command for monitoring resource usage statistics. This command calls `docker stats` and returns a live data stream for running containers. [1]

In the output fragment, you can see an example of `docker stats` for several running containers.

```
$ docker stats
```

CONTAINER ID	NAME	CPU %	MEM USAGE / LIMIT	MEM %	...
28fa294fa8fe	demo1	0.00%	4.523MiB / 1.939GiB	0.23%	...
b7cbbd9b0282	demo2	1.20%	92.71MiB / 1.939GiB	4.67%	...
04e48f046961	demo3	0.03%	13.14MiB / 1.939GiB	0.66%	...
1768815caff8	demo4	0.46%	376.8MiB / 1.939GiB	18.98%	...
3a6b3d036d3e	demo5	0.03%	10.48MiB / 1.939GiB	0.53%	...

■ **Listing 1.1** `docker stats` command output example [1]

Since not all statistics fit in the figure, let us list them. This command provides:

- CPU % – host’s CPU percentage, used by a container.

- MEM % – host’s memory percentage, used by a container.
- MEM USAGE / LIMIT – total and allowed amounts of memory usage.
- NET I/O – The data amount that the container has sent and received over its network interface.
- BLOCK I/O – The data amount that the container has read to and written from block devices on the host.
- PIDs – Processes or threads number that the container has created.

This method provides basic data and metrics, which may be sufficient. There are two remaining methods for more detailed statistics.

1.3.3 Monitoring via pseudo-files

This method is based on the control groups[6], which are used by Linux containers to track groups of processes. They also provide information about CPU, memory, block I/O usage, and network usage. The control groups are exposed through a pseudo-filesystem that you can find under `/sys/fs/cgroup`. There are multiple sub-directories. Each of them corresponds to a different group hierarchy.

Below you can see an example of the contents of the `memory.stat` pseudo-file provides information about memory usage within a given operating system.

```
$ cat /sys/fs/cgroup/memory/docker/$CONTAINER_ID/memory.stat

cache 532480
rss 10649600
mapped_file 1576960
writeback 0
swap 0
pgpgin 302242
pgpgout 296556
pgfault 1142200
pgmajfault 125
inactive_anon 16384
active_anon 577536
inactive_file 11386880
active_file 11309056
unevictable 0
hierarchical_memory_limit 18446744073709551615
hierarchical_memsw_limit 18446744073709551615
total_cache 22798336
total_rss 491520
total_rss_huge 0
total_mapped_file 1576960
total_writeback 0
total_swap 0
...
```

- **Listing 1.2** `memory.stat` pseudo-file content example [2]

It should be noted that one of the benefits of this method is speed. It is the fastest way to get metrics. This method also provides sufficient information about CPU and memory metrics. However, this method has some limitations on I/O and network metrics. [7]

1.3.4 Monitoring via API

This is another method of Docker monitoring. As the `docker stats` command method, it also continuously reports a live stream of the metrics. The difference between API and stats command methods is that the API provides more details about metrics. But basic metrics groups CPU, memory, I/O, and network are still the same.

By default, the API can be accessed by the Unix domain socket. It listens on the following socket:

```
unix:///var/run/docker.sock
```

Docker provides REST API for communication. This API can be used not only for container monitoring but also for container management. It offers container creation, file system management, launch, and others. A more detailed description of the API. [3]

To get the required metrics, docker provides the following endpoint:

```
GET /containers/(id or name)/stats
```

It provides data in JSON format. Below is an example of a GET request sent to this endpoint.

```
{
  "read" : "2015-01-08T22:57:31.547920715Z",
  "networks": {
    "eth0": {
      "rx_bytes": 5338,
      "rx_dropped": 0,
      "rx_errors": 0,
      "rx_packets": 36,
      "tx_bytes": 648,
      "tx_dropped": 0,
      "tx_errors": 0,
      "tx_packets": 8
    },
    "eth5": {
      "rx_bytes": 4641,
      "rx_dropped": 0,
      "rx_errors": 0,
      "rx_packets": 26,
      "tx_bytes": 690,
      "tx_dropped": 0,
      "tx_errors": 0,
      "tx_packets": 9
    }
  },
  "memory_stats" : {
```

■ Listing 1.3 Docker API request example [3]

As we can see, this method is comparable in terms of the completeness of metrics with monitoring via pseudo-files. However, this method, like the pseudo-files method, has some limitations on I/O metrics. [8]

1.3.5 Docker monitoring summary

So we have described three ways that Docker provides for collecting metrics. Of course, the choice of method should be based on your needs. A large amount of system metrics is a merit, but the difficulty of implementation can be another important aspect to consider.

1.4 Bare metal monitoring

This section will focus on what metrics the operating system provides and how we can get them. For this section, we will limit ourselves to Linux-based operating systems, assuming that this kind of system will be used in our solution.

The ways in which bare metal monitoring is realized are similar to the Docker ones. Linux provides two ways to get metrics: via commands and pseudo-files.

1.4.1 Monitoring via commands

The easiest way to find out the metrics in Linux is to use the commands. Linux provides a wide range of commands with which you can get metrics. For example, you can get memory usage info via `free` command. Below you can see an example of calling this command.

```
$ free -h
```

	total	used	free	shared	buffers	cached
Mem:	1.9G	1.7G	293M	122M	41M	635M
-/+ buffers/cache:		1.0G	970M			
Swap:	1.0G	290M	733M			

■ **Listing 1.4** free command calling example

As you can see, this is a very concise output. This method is human-acceptable and provides basic information. For more details, let's look at the following monitoring method.

1.4.2 Monitoring via pseudo-files

This method is based on the pseudo file system `proc`, located in the `/proc` path in the Linux filesystem. This pseudo-file system provides information about running Linux systems.

There are specific entries in the `/proc` that provide kernel metrics. For example, `/proc/meminfo` gives you information about the distribution and utilization of memory. Among other things, you can find metrics such as the available, active, and total volume of RAM you can use. You can see an example of memory pseudo-file content below:

```

$ cat proc/meminfo

MemTotal:      2033396 kB
MemFree:       300980 kB
MemAvailable:  783048 kB
Buffers:       42776 kB
Cached:        650564 kB
SwapCached:    28716 kB
Active:        575916 kB
Inactive:      995556 kB
Active(anon):  226788 kB
Inactive(anon): 768532 kB
Active(file):  349128 kB
Inactive(file): 227024 kB
Unevictable:   0 kB
Mlocked:       0 kB
SwapTotal:     1048572 kB
SwapFree:      750924 kB
Dirty:         0 kB
Writeback:     0 kB
AnonPages:    851552 kB
Mapped:        216500 kB
Shmem:         125092 kB
KReclaimable:  74412 kB
Slab:          112424 kB
SReclaimable:  74412 kB
SUnreclaim:   38012 kB
...

```

■ **Listing 1.5** Memory pseudo-file content example

The pseudo file system `proc` provides a comprehensive set of metrics. Among other information, you can find the amount of data transferred over the network, the use of the processor, memory, and others. More information can be obtained from the following source: [9]

1.4.3 Bare metal monitoring summary

So we have described two ways that Linux provides metrics. Of course, the choice of method should be based on your needs. A large amount of system metrics, of course, is a merit, but sometimes a large amount of information can be related to complex implementation.

1.5 Monitoring systems

Using the knowledge gained in the previous chapters, we could design a solution that used one of the described methods to obtain metrics. However, instead of implementing this solution on our own, we should pay attention to existing services and components solving similar problems. Integrating such a component could reduce the complexity of the whole monitoring solution and make the solution more general.

1.5.1 cAdvisor

In this section, I will provide information about cAdvisor. I will describe the functionality that cAdvisor offers as well as some of its disadvantages.

cAdvisor functionality

cAdvisor stands for container advisor. It is an open-source tool that allows you to monitor running Docker containers. cAdvisor collects, aggregates, and processes information about each resource's usage and performance characteristics. [10]

It also provides a web UI to visualize collected metrics that use previous data. However, it is worth noting that cAdvisor doesn't have persistent storage, and these metrics are real-time only.

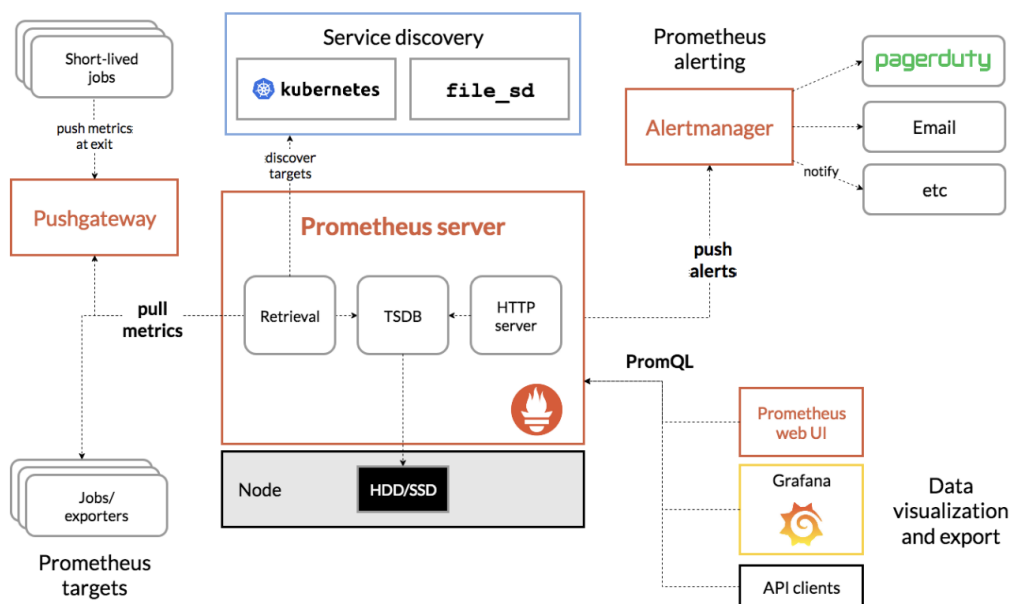
However, cAdvisor can also be combined with other solutions. Besides web UI, cAdvisor also provides a REST API, which allows access to raw and aggregated metrics. Thanks to this, cAdvisor is a rather interesting component of the future solution.

1.5.2 Prometheus

In this section, we will explore the capabilities of Prometheus in detail and gain an understanding of its architecture. We will also look at the integration capabilities of the components that Prometheus supports.

Prometheus is an open-source solution that provides systems monitoring and alerting. Instead of cAdvisor, which provides metrics about running containers, Prometheus works with more general metric representations. For Prometheus, metrics are time series data, which may differ from application to application. [11]

Let us study the architecture of Prometheus and see its components. The following figure shows the diagram that illustrates the Prometheus architecture and some related components:



■ **Figure 1.2** Prometheus architecture [11]

Let us highlight and describe the following parts of this ecosystem:

- Exporters
- Prometheus server
- Data export

Exporters

Under the term exporter, we will imagine a library or service that helps export existing metrics from third-party systems such as the Prometheus metrics. There are a lot of such exporters; however, we will be interested in the Docker and bare metal metrics exporting. For a more detailed study of exporters, please read the official documentation. [12]

Some systems provide the export of metrics in Prometheus format without the need for an additional exporter. Luckily for us, Docker is one such system. Exporting data in Prometheus format is an additional option for the Docker daemon command. If you start the command `dockerd` with the additional option `--metrics-addr` and specify an address and port like this `--metrics-addr 127.0.0.1:9323`, it allows you to get Docker metrics in Prometheus format at `127.0.0.1:9323/metrics`.

This option for the `dockerd` command is experimental; therefore, the daemon must run in experimental mode. It was marked as experimental because metrics and metric names could still change. This particular point must be considered when a solution is designed. [13]

Prometheus server

The Prometheus server is the main component of Prometheus architecture and does the actual monitoring work. The Prometheus server is divided into three components:

TSDB stands for Time Series Database. This component is responsible for metrics data storage.

It stores the data retrieved from Data Retrieval Worker. This component uses persistent storage to store data. This storage does not have to be local, and it also supports remote storage systems. It is also necessary to mention that these data are collected in a specific format. This means that you cannot use a relational database to store them.[14]

Data Retrieval Worker is responsible for pulling metrics from target resources, which are services, applications, etc, and then it pushes this data to the TSDB component. Metrics pulling is realized using exporters or over endpoints. The last method you can see in the Docker metrics exporting process is when the Docker daemon creates the `/metrics` endpoint.

HTTP Server is a web server component that is used for displaying data to Prometheus Web UI or some other visualization tools, like Grafana. Communication with HTTP Server is realized using PromQL. PromQL is a special functional query language provided by Prometheus.[15]

In this section, we have described the key points of the Prometheus server. However, it is worth supplementing information about PromQL.

Data export

In the previous section, we noted that the data is exported from the HTTP Server using PromQL. In this section, we will analyze the PromQL in more detail to understand how to process the data. This intro is necessary to understand the structure of requests and the communication process. Here we will describe the PromQL basics. For more information about PromQL, please read the documentation. [4]

PromQL supports the evaluation of four types of expressions:

- The Instant vector is an expression that returns, depending on the metric, a set of time series containing a single sample for each time series.
- The Range vector is an expression that returns, depending on the metric, a set of time series containing a range of data points over time for each time series.
- Scalar is a simple numeric floating point value.
- String is a simple string value.

It is necessary to mention that the String type is currently unused so that we will focus on the others. [4]

Instant vector selectors allow you to select a set of time series for a single sample value at a given timestamp. In the simplest form, only a metric name is specified. It returns the vector containing elements for all time series with the same metric name.

This example selects all-time series that have the `http_requests_total` metric name:

```
http_requests_total
```

- **Listing 1.6** Instant vector selectors example [4]

Range Vector Selectors work similarly to instant vector selectors. The difference is that they select a range of samples back from the current instant. There is also specified a time duration that is appended in square brackets (`[]`) at the end of a selector. It specifies how far from now values should be fetched.

In the following example, we select all the values we have recorded within the last 5 minutes for all time series that have the metric name `http_requests_total` and a job label set to Prometheus:

```
http_requests_total{job="prometheus"}[5m]
```

- **Listing 1.7** Range Vector Selectors example [4]

Float literals This type describes a numeric value. It can be an integer or floating-point number in the following format:

```
[ -+ ] ? (
    [ 0 - 9 ] * \. ? [ 0 - 9 ] + ( [ e E ] [ - + ] ? [ 0 - 9 ] + ) ?
    | 0 [ x X ] [ 0 - 9 a - f A - F ] +
    | [ n N ] [ a A ] [ n N ]
    | [ i I ] [ n N ] [ f F ]
)
```

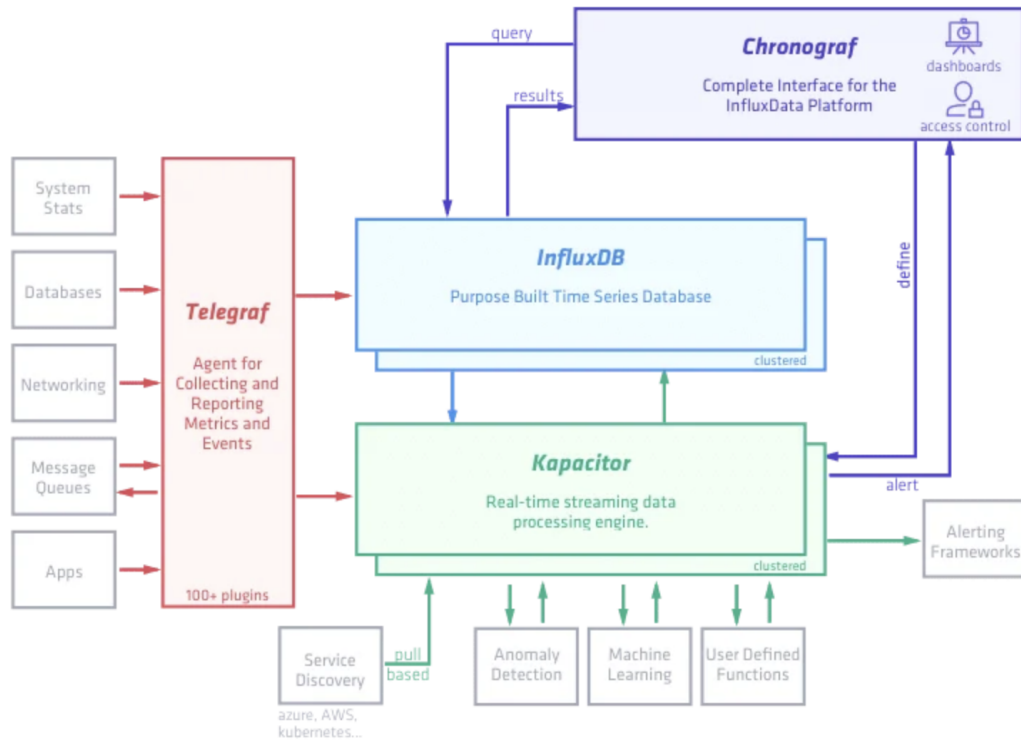
- **Listing 1.8** Float literals example [4]

1.5.3 TICK stack

TICK stands for Telegraf, InfluxDB, Chronograf, and Kapacitor. TICK stack is an open-source monitoring solution that combines components for collecting, storing, visualizing, and manipulating any time series data.

Let us look closer at each of the components, describe the principle of their work, and discuss the interaction of components with each other.[16]

In the following figure, you can see the diagram that illustrates the TICK stack architecture:



■ **Figure 1.3** TICK stack architecture [17]

Telegraf

Telegraf is a component for collecting and reporting metrics. Its main task is collecting information and providing it with some storage solution. Telegraf is a plugin-driven solution. It means that the main component of Telegraf can be easily connected with many different plugins, which are typically input and output plugins. [18]

Output plugins are focused on data storage. Supported output solution is not only InfluxDB but many other solutions such as Prometheus, Graphite, Kafka, AMQP, and others. [19]

Input plugins are focused on collecting metrics. Supported inputs are Docker, Spark, AWS, and others. There are also a lot of bare metal monitoring plugins, that we will use. Of course, the most interesting plugin for us would be the Docker plugin. The principle of how this plugin gets metrics from Docker was described earlier. It is the monitoring via the API principle [20].

InfluxDB

InfluxDB is a time series database. Together with Telegraf, they provide similar functionality as the Prometheus server. For data interaction, InfluxDB offers its query language. Instead of PromQL, it is a SQL-like language. It is called InfluxQL. [21]

Chronograf

This is another component of the TICK stack. It provides functionality to visualize data from InfluxDB components and define rules for Kapacitor to send alerts. Let us take a closer look at the functionality that Chronograf provides.

Visualization Chronograf is a complete visualizing solution. It allows you to create dashboards from scratch or customize templates. The Chronograf pre-created dashboards for over 20 apps, including docker, Kubernetes, Apache, and others.

Kapacitor's API Chronograf provides an API for InfluxDB data processing by Kapacitor. It allows Kapacitor to create alerts, detect anomalies in data using different approaches, and run ETL jobs. Alerts creation is supported for Slack, Telegram, and other targets.

It also supports database management, providing some security options, and managing retention policies. Thus, Chronograf is the user interface and administrative component of the InfluxDB platform. [22]

Kapacitor

Kapacitor is a native data processing engine for InfluxDB. It also provides some other features[23]:

Alerting Kapacitor provides functionality for alerting using the data obtained from Chronograf. Alerting is designed as the publish-subscribe pattern.

Data processing Kapacitor also provides functionality for processing streaming data. Kapacitor allows you to process data before and after saving it to InfluxDB. Thus, with the help of Kapacitor, it is possible to carry out preprocessing and postprocessing of data, allowing them to be used for analytics.

Anomaly detection Kapacitor has a control system that allows you to call custom functions and also provides an interface that allows it to be integrated with anomaly detection mechanisms, such as machine learning libraries. It helps to automate the entire system.

1.5.4 Monitoring systems summary

In this section, we considered several systems for monitoring. Let us summarize the analysis of these systems.

Obviously, cAdvisor cannot become a standalone solution due to the lack of persistent storage. However, it provides a convenient API that other components can use. Therefore, it can be useful within any solution as a separate component.

Unlike cAdvisor, Prometheus is a more complex and general solution with many more use cases. With persistent storage and the ability to work with more than just Docker metrics, Prometheus can become a crucial component of the monitoring solution.

In the case of the TICK stack, it is a complex and general solution with many more use cases. It also, as Prometheus, can be used as a key component of the future solution.

Thus, we find two suitable designs based on which a future solution can be developed.

Next, we will analyze the solution requirements, such as communication security, data consistency between the collection and monitoring solutions, prediction of failure on the monitoring solution side, and others.

We will also find out if both systems meet our requirements and proceed to our solution's design.

1.6 Communication security requirements

In this chapter, we will look at ways to make communication between two applications secure. The following requirements must be met to ensure secure communication between two applications[24]:

- Authentication: The process of identification for both subjects of communication.
- Privacy: Guarantee that no one except the recipient will read the data.
- Integrity: Guarantee that the data will not be changed in transit.
- Non-repudiation: Mechanism to prove the action of sending data by the sender.

In this diploma work, we will focus on the first three requirements. In the case of Non-repudiation, this is due to specific communication. In our case, communication is made between two internal systems. It means there is no need to prove it in case the receiver and the recipient are identical.

1.6.1 Authentication

This subsection is devoted to authentication. It aims to describe the possible methods of authentication and potential problems associated with it and to find a suitable solution to these problems.

Authentication and Authorization

Authentication and authorization are common terms in the world of identity and access management (IAM)¹. While these terms may sound similar, they are both distinct security processes, and understanding the difference between them is key to successfully implementing an IAM solution.

Authentication is the act of confirming that users are whom they say they are. Passwords are the most common authentication factor – if the user enters the correct password, the system assumes the identity is valid and grants access.

Authorization in system security is the process of granting a user permission to access a specific resource or function. This term is often used interchangeably with access control or client rights. A good example is giving someone permission to download a certain file on a server or giving individual users administrative access to an application. [25]

HTTP authentication

HTTP supports using several authentication mechanisms to control access to pages and other resources. All of these mechanisms are based on using a 401 status code and the `WWW-Authenticate` header in the response.

¹Identity and Access Management – a set of approaches, procedures, technologies, and special software tools for managing user credentials.

The most commonly used HTTP authentication mechanisms are[26]:

Basic authentication is the principle when the client sends the username and password as plain base64 encoded text. It should only be used with HTTPS, as the password can be easily captured and reused over HTTP.

Digest authentication is the principle when the client sends the hashed password to the server. Although the password cannot be captured over HTTP, it is possible to replay requests using a hash of that password.

NTLM is used as a secure request/response mechanism to prevent password capture or HTTP replay attacks. However, authentication is connection-specific and will only work for persistent HTTP/1.1 connections. Because of this, it may only work on some HTTP proxies and may introduce a large amount of two-way delay if connections to the web server are regularly closed.

Passwords are not secure and should not be used for authentication alone. They are hard to remember, so users are tempted to use weak passwords and reuse them on multiple sites. Even if a password is strong, it's still just a short string that users know. We will try to find other authentication mechanisms that are considered more secure. [27]

Token assisted authentication

This authentication method is most often used when creating distributed Single Sign-On (SSO) systems, where one application (service provider) delegates the user authentication function to another application (identity provider).

The implementation of this method is that the identity provider provides reliable information about the user in the form of a token, and the service provider application uses this token to identify, authenticate, and authorize the user.

Several standards precisely define the protocol of interaction between clients and identity provider and service provider applications and the format of supported tokens. The most popular standards include OAuth, OpenID Connect, SAML, and WS-Federation.

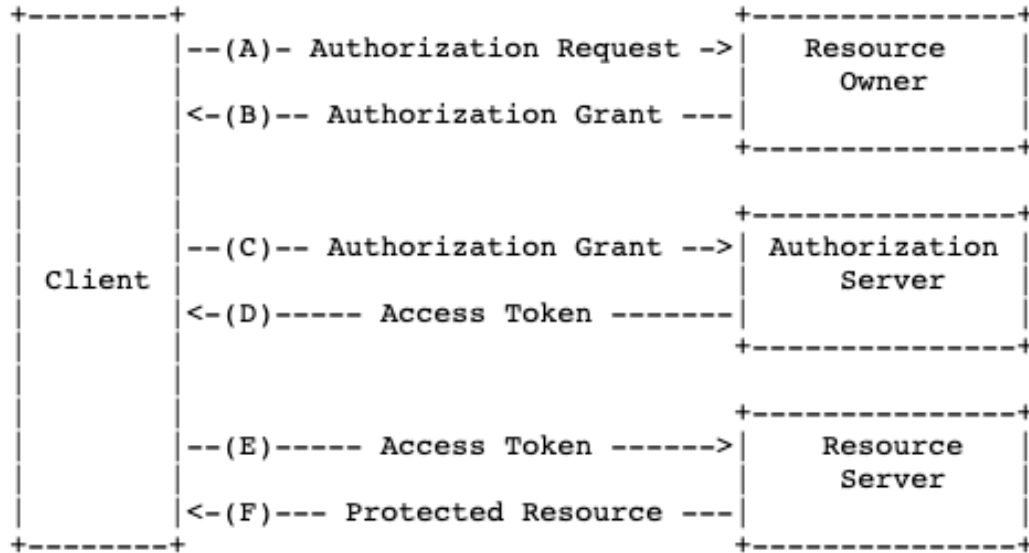
OAuth and OpenID Connect standards

OAuth is an authorization protocol that allows a third party to grant limited access to a user's secure resources without providing credentials: username and password. The current version of the protocol is OAuth 2.0, which was published in 2012. [28]

Four roles are defined in the OAuth protocol documentation:

- Resource Owner – an entity that can provide access to secured resources.
- Resource Server – a server on which secure resources are deployed.
- Client – an application that requests access to secure resources from the resource owner's credentials.
- Authorization server – a server that issues access tokens to the client after successful authentication of the resource owner and access permission.

For a complete explanation, we describe the interaction between these roles. Interaction can be easily divided into the following steps, which are shown in the following figure:



■ **Figure 1.4** OAuth protocol flow [28]

- (A) The client sends a request to authorize resource owners. The request can be sent directly to the owner or through an authorization server, which in this case, will play the role of an intermediary.
- (B) The client receives an authorization token that represents the resource owner's authorization.
- (C) The client requests the authorization server for an access token based on the obtained authorization permission.
- (D) The authorization server identifies the client and checks that it has valid authorization. If so, it issues an access token to the client.
- (E) The client requests the resource server for secure resources based on the access token.
- (F) The resource server checks the validity of the token and processes the request.

Next, we will describe the OpenID protocol.

OpenID Connect (OIDC) is an authentication protocol based on the OAuth 2.0 family of specifications. It uses a simple JSON Web Token (JWT) that you can get using OAuth 2.0 compliant flows.

While OAuth 2.0 is about resource access and sharing, OIDC is mainly about user authentication. Its purpose is to provide you with a single login for multiple sites. Every time you need to sign in to a website using the OIDC protocol, you will be redirected to the OpenID website you sign in to and then back to the website.

Authentication using a certificate

Client certificate authentication is the process of how users securely access a server or remote computer by exchanging a digital certificate. A digital certificate is considered a digital ID. It is used to cryptographically bind the identity of a customer, employee, or partner to a unique digital certificate (typically including the name, company name, and location of the owner of the digital certificate). The digital certificate can then be mapped to a user account and used to control access to network resources, web services, and websites. [29]

Certificate authentication offers a higher security standard than password authentication can. Federal organizations often use it. Certificate authentication has been successfully used in defense, healthcare, and banking areas. These organizations often consider the PKI infrastructure to be strategically important to their security goals.[30]

Certificate authentication can be inappropriate for two reasons. Client authentication involves a non-trivial registration process on the part of the client. The certificate is produced on one device, so if the client wants to log in from another device, they will need to transfer it securely, which is also hard to guarantee. [31]

The stated disadvantages are not particularly important for us since these processes are simplified when communication occurs within the same system.

Authentication summary

In this chapter, we got acquainted with the main principles of authentication. In our case, the advantages of authentication using certificates are obvious when it comes to an internal application, while the disadvantages are not so significant. Undoubtedly the most suitable for our case is authentication with a certificate.

1.6.2 Privacy and Integrity

This section will look at two important requirements for communication security between two applications. We will also analyze the difference between these requirements and determine what is needed to fulfill them.

Privacy

The main purpose of privacy in case of communication is to protect data against using it by third parties. The most common way to protect data is encryption. Secure Sockets Layer (SSL) is the de facto Internet standard for encrypting communication over the Internet. It is more recently known as Transport Layer Security or TLS. [32]

In essence, it is a cryptographic protocol that provides end-to-end protection for data sent between applications over the Internet. By encrypting data in transit, we will fulfill the Privacy requirement in relation to communication between applications.

Integrity

The main purpose of integrity is to ensure that the data are not changed in transit. The transmitter accompanies the data with code to solve this issue. This code is known as the Message Integrity Code (MIC). The point is that MIC is generated by the way, which knows only the transmitter and recipient, and only they can reproduce this code. Thus when the receiver matches the MIC provided by the transmitter, the data will be considered authentic.

Discussion

Data integrity is different from data privacy. However, both solve similar tasks – data protection. In integrity data transmission, the data is encrypted before transmission so that only the intended recipient will be able to recover the data. Thus private data can not be recovered by hackers, and integrity is carried out in this case. However, the data may still have been changed in transit. So it is recommended to accompany the private data with a MIC, but this is not required. On the other hand, data can be transmitted unencrypted but accompanied by a MIC. In this case, the data is not confidential, but its authenticity can be verified. [33]

1.6.3 Security conclusion

This chapter taught us the main principles of secure communication between applications. Certainly, security is a very important requirement for our solution. We will meet these requirements as well as possible in design and project implementation.

1.7 Data and Metrics visualization

One of the requirements is to visualize data and metrics that will be sent from the monitoring solution to collection one. Compared to others, this requirement is very concrete since the technology chosen by the customer is Grafana. Let us get acquainted with this technology, find out its capabilities, and identify the problems that we may encounter when integrating it into our solution.

1.7.1 Grafana

Grafana is a database analysis and monitoring tool. It allows you to create dashboard visualizations of key metrics that are important to you.

Grafana supports a huge number of data sources, including Prometheus, InfluxDB, and many others. [34] It also supports visualization for DBMS such as MySQL, PostgreSQL, and others. So it can be perfectly integrated into our solution.

Grafana also provides alert-sending functionality. It can notify you about problems in your system. Thus, Grafana is an important component of the system. It is not only an easily integrated component that provides visualization but is also able to take part in the failure prediction process.

Let us also describe alternative solutions that are parts of monitoring systems and describe some of the features that they offer.

1.7.2 TICK stack components

It is worth paying attention to the fact that the Chronograf, paired with the Kapacitor, offers similar functionality as the Grafana. Some important features of both solutions should be noted.

Chronograf and Kapacitor only work within the TICK stack and are, therefore, much easier to use when using InfluxDB. At the same time, they offer more options than Grafana.

Quite important can be: connecting third-party solutions for failure prediction, preprocessing and postprocessing of data, running ETL jobs, and others. It is undoubtedly a big benefit for the future solution.

1.7.3 Prometheus

In the case of Prometheus, it is proposed to use Grafana for visualization. [35] It simplifies the choice of platform for visualization.

Prometheus also offers to use its system to send alerts. It is called Alertmanager. It provides functionality for grouping and routing. It also takes care of deduplication and recipient integration.

Thus, Prometheus does not provide additional functionality but only improves alert management.

1.7.4 Visualization summary

In this section, we got acquainted with the solutions offered by monitoring systems already familiar to us. We also learned that the TICK stack offers much more functionality than Grafana.

However, it is also important for us that we work not only with metrics but also with data. And Grafana's support here for a large number of resources is a big plus. Since the use of Grafana is one of the requirements, we will use it for our solution. However, with the increase in the scale of the solution itself, it was appropriate to consider using CK components of the TICK stack.

1.8 System failure prediction

It should be noted that this topic is very broad. There are the following approaches for failure prediction:

- Thresholds method
- Forecasting methods

Let us look at some of them.

1.8.1 Thresholds method

The method is based on setting thresholds for different metrics. Based on this threshold, we identify and prevent the problem. To implement this method, you must take the following steps[36]:

Select metrics Metrics are numerical data you can measure on the IT element directly. For example, very informative capacity metrics. They indicate that your unit is running low on the capacity for some resources, which can be disk, memory, CPU, transaction processing capacity, and others.

Set thresholds Once you have defined the metrics, detection and prediction come down to choosing the suitable thresholds for your metrics. The threshold value can indicate that a failure is occurring or indicate a problem is occurring. In other words, it realizes the detection and prediction of failures.

Thresholds for detection are usually easy to determine. The normal range of key metrics is usually known, and key failure events are also known. For example, more than 95% CPU usage indicates a server capacity issue.

Thresholds for prediction involve a trade-off. There is a trade-off between getting a false prediction and not having enough early warning of a problem. You can set the free space threshold for disk space from 70% to 90%.

Fine-tune the thresholds based on historical data The essence of this point is to adjust the threshold depending on the analysis of historical data and ensure the best work of thresholds.

This simple method is very popular and often used in practice. The essence of the method, however, for the most part, is not the prediction but the detection of failures. However, good tuning can cope with both tasks to some extent.

1.8.2 Forecasting methods

There are a wide variety of forecasting methods. The most popular groups of them are exponential smoothing methods and ARIMA models. Let's study the most prominent representatives of these groups.

Holt-Winter's seasonal method

Holt-Winters Exponential Smoothing is used for forecasting time series data. This method is a member of the exponential smoothing methods group. It takes into account both the trend and seasonal variations. This method is based on the Exponential Smoothing method, which is a *“technique forecasts the next value using a weighted average of all previous values where the weights decay exponentially from the most recent to the oldest historical value.”*[37]

Holt-Winters Exponential Smoothing corrects the shortcomings of the Exponential Smoothing method by taking into account the trend and seasonality when working with data.

The importance of this method for us lies in the fact that it is natively supported by InfluxDB and can be easily used in our solution. [38]

Non-seasonal ARIMA models

ARIMA stands for AutoRegressive Integrated Moving Average. The non-seasonal ARIMA model contains two other models.

For a more accurate understanding, it is necessary to explain these two models. The autoregression model predicts the value using the linear combination of the past, and it is the reason why it is called autoregression. The moving average model predicts the value using past forecast errors in a regression-like model.

The non-seasonal ARIMA model is a combination of differencing with autoregression and a moving average model. [39]

Unfortunately, this algorithm is not natively supported by either Prometheus or TICK stack, despite this, it is a very popular forecasting way, and there are undoubtedly ways to integrate it.

1.8.3 System failure prediction summary

In this section, we reviewed the most popular methods for System failure prediction. In order to discuss how effective these methods will be for our solution, it is necessary to have some statistics about how well these methods do in our case. In this work, we do not aim to create a model that detects System failure on time. This is a complicated task that is rarely achieved in practice.

In this work, we will find a compromise between the complexity of implementing a solution for System failure prediction and the timeliness of detection failures.

1.9 Data consistency

In our case, we are talking about consistency between more databases over time. It means for us that data needs to be synchronized correctly over time. Let us introduce the reader to the topic of data synchronization.

Database synchronization is the process of establishing data consistency across multiple databases. Often this means automatically copying changes in both directions.

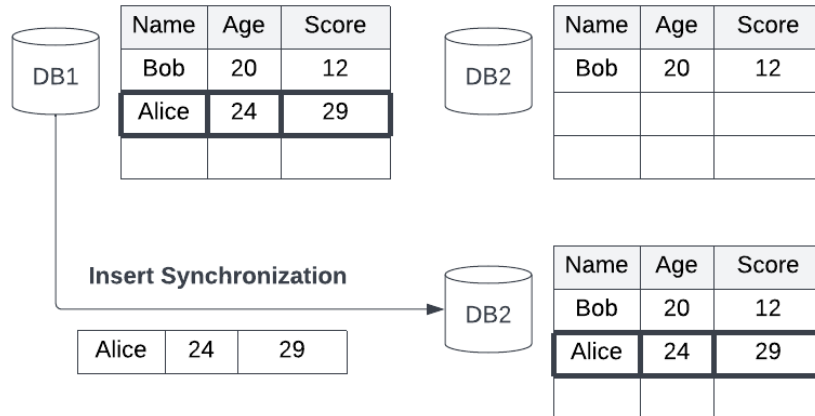
Data reconciliation over time should be continuous. The most trivial case of synchronization is pulling data from the source database to the target. [40]

Based on the requirements analysis, we can communicate only in one direction. It means that synchronization between databases can only be one-way. Given this requirement, let us focus on the types of one-way synchronization. There are usually three types of synchronization that you have to deal with:

- Insert synchronization
- Update synchronization
- Delete synchronization

1.9.1 Insert synchronization

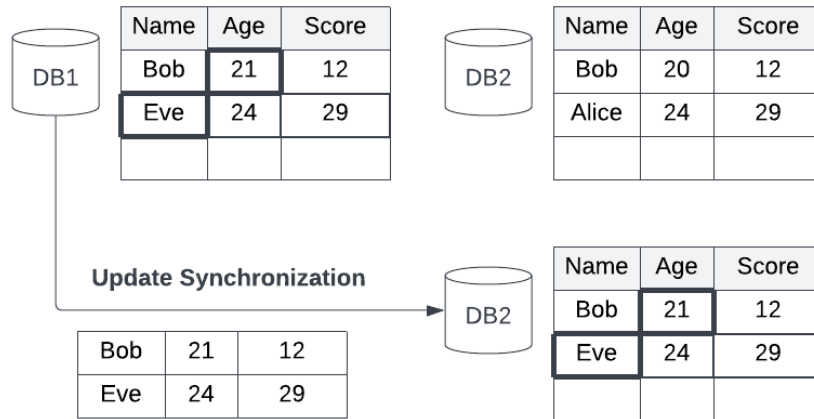
This process proceeds as follows, new records from the source table will be automatically transferred to the target table over time. If there are no matching records in the target table, the synchronization process will insert the missing records into the target tables.



■ **Figure 1.5** Insert synchronization example

1.9.2 Update synchronization

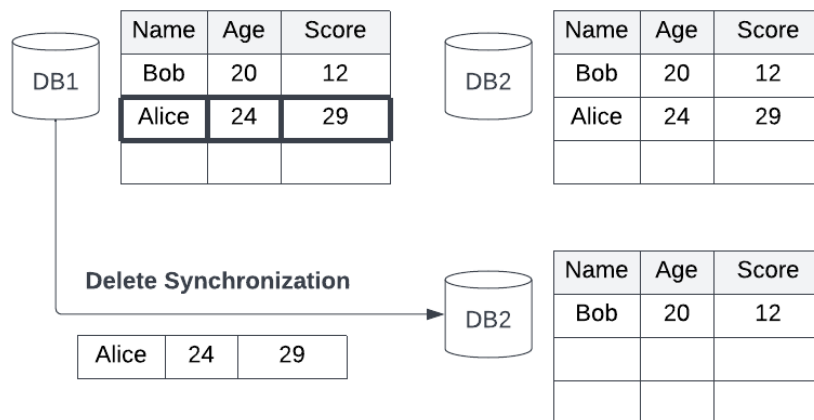
This process proceeds as follows. When changes occur in the source database, the corresponding changes must be made in the target database. The process takes place in two stages. First, the values of the records in the target and source databases are compared. The target table is then changed from the source table. As a result, your data is synced after the update.



■ **Figure 1.6** Update synchronization example

1.9.3 Delete synchronization

The delete synchronization process is as trivial as the insertion synchronization process. If some records have been deleted from the source database, the corresponding records must also be deleted from the target database. This way, both databases will be synchronized after deletion.



■ **Figure 1.7** Delete synchronization example

1.9.4 Data consistency summary

In this chapter, we have covered the necessary information regarding synchronization between multiple databases. We also described the types of synchronization that can occur during one-way communication. We will apply the knowledge gained in this chapter to design a future solution.

1.10 Technology Analysis

We are limited in the choice of technologies and languages since we need to fulfill the requirement associated with JVM 11. But still, several languages fall under this requirement: Scala, Kotlin, Groovy, Clojure, and of course, Java.

We will not conduct a comparative analysis of all languages. Instead, we will limit ourselves to the two most popular languages according to the PYPL version as of September 2022 [41]. Namely, we will compare the use of Java (#2) and Kotlin (#12).

In addition to choosing a language, we should decide on the framework and build an automation tool.

In the case of a framework, Spring is the most popular solution. Let us find out if it suits us and analyze why we should use it.

As for the build automation tool, there are two most popular options: Apache Maven and Gradle. Let us compare them and find the most suitable option for us.

1.10.1 Java & Kotlin comparison

In this section, we will discuss the similarities and differences between the two languages and then decide which one is best for us.

Similarities

The two languages have some similarities. They are even interoperable. This means that Java and Kotlin files can coexist in the same project or JAR package. Let us explore these similarities in more detail.

Static typing Java and Kotlin have statically typed languages. It means that type checking is going on in compile time. (There are also dynamically typed languages such as PHP, Python, and JavaScript.)

Bytecode Both languages translate the code into bytecode that is executable by the JVM.

Kotlin usage advantages

It is important to note that Kotlin is a much younger programming language, so it solved several problems in comparison with Java. [42]

Extension Functions One of the interesting solutions of Kotlin is the extension function. In Java, extend methods require inheriting the class and overriding the method, but thanks to the extension function. It is not necessary, and you can extend methods without inheriting the class.

Conciseness Kotlin is a more concise language than Java. It greatly reduces boilerplate code and results in fewer lines of code than Java, making the syntax more straightforward and readable. This important quality makes the code easier to read and less prone to human error.

Null Safety Kotlin offers null safety, unlike Java. Java allows you to assign null values to objects. Therefore, when accessing a member of an object with a null value, a `NullPointerException` is thrown, which is fairly common in Java. Kotlin does not allow default values to be null and therefore provides more of a stability code.

Smart Casting In Kotlin, thanks to the smart casting compiler automatically manage types. In Kotlin programmer does not need to check the variable type accordingly per operation, unlike in Java.

Functional Programming Kotlin offers to use not only OOPs behavior but the behavior of Functional programming languages. Unlike Kotlin, Java is an Object-Oriented Programming language only.

Well, let us move on to the advantages of Java.

Java usage advantages

The main advantages of Java stem from the fact that it is an extremely popular programming language. From here, follow the next advantages: [43]

Detailed documentation Java is one of the most popular programming languages in the world. Thus, its popularity has been maintained for a very long time. Also, Java is often used to create enterprise projects for which good documentation is very important. As a result of which, Java is a language with detailed documentation. At the same time, Kotlin is a relatively young programming language that cannot yet boast of this.

Third party libraries This point is similar in its argumentation to the previous one. It is also caused by the incredible popularity of Java and its use in the enterprise environment. Because of this, Java has a massive array of third-party libraries compared to Kotlin. It is not a big advantage since Kotlin is almost completely compatible with Java, and libraries for Java can be used in projects written in Kotlin.

We have considered the pros of choosing both languages. Let us make a conclusion and choose the most suitable one for us.

Summary

Each of these languages is popular and is used by a large number of developers around the world. Using both Java and Kotlin certainly has its advantages. The advantages of Java are especially relevant for large projects in which it is necessary to use a large number of third-party libraries that would have good documentation.

The scale of our implementation is not so large, and the advantages of Java are not so significant for us. On the contrary, Kotlin's features can simplify development and reduce the complexity and time of writing code, which is appropriate in our situation. Therefore, in this case, we prefer Kotlin.

1.10.2 Maven & Gradle comparison

In this section, we will discuss the similarities and differences between the two tools and then decide which one is best for us.

Maven

Apache Maven is a powerful build automation tool to manage projects and maintain dependencies. Maven uses pom.xml for project configuration.

Gradle

Gradle is a build automation tool used in software development to project build automatization and dependencies management. Gradle has its DSL stands for domain-specific language based on Groovy or Kotlin code.

Maven usage advantages

Let us provide information about what advantages Maven has over Gradle.

Plugins One of the main advantages of Maven is the management and creation of plugins. Creating plugins in Maven is easy, and there are hundreds of plugins Maven offers.

Documentation Maven has a more extended history of use compared to Gradle. Thanks to this, you are unlikely to find a project in the JVM world that would not have a way to add Maven dependencies but would have it in Gradle, while the reverse situation can be. Also, the experience of using Maven is greater, and the experience of current and past users can be very valuable for solving complex problems.

Now that we have seen the advantages of using Maven, it is good to suggest taking a look at the advantages of Gradle.

Gradle usage advantages

Let us provide information about what advantages Gradle has over Maven.

Performance Gradle introduced several performance optimizations missing from Maven to improve build performance.

Conciseness Maven is much less concise because it uses XML for its configuration. Gradle, as already mentioned, uses DSL and boasts conciseness.

User experience Gradle is more modern and more end user-oriented. Moreover, although Gradle has a higher entry threshold than Maven, Gradle provides more convenience and variability in use. Maven, as the number of dependencies, grows and the pom.xml grows, becomes clumsy, and this harms the user experience.

Summary

In conclusion, I would like to note that both technologies are a good choice for your project. Maven is a simpler technology with a low entry threshold, Gradle, in turn, is more complex, but it also greatly simplifies working with the project in the future. For us, perhaps Gradle is preferable based on the listed qualities. Therefore, we will use it in our solution.

1.11 Analysis summary

This chapter has taught us all the necessary material to create a future solution. In the future, based on the knowledge gained, we will create the solution's architecture and solve the aspects of synchronization, security, and failure prediction. Also, the received material will allow us to choose the suitable technology for monitoring the hospital environment, which is one of the key issues of this work.



Chapter 2

Design

In this chapter, we will focus on designing the final solution. Our attention will be given to two solutions based on two technologies taken as the basis for collecting metrics. Next, a discussion will be held, as a result of which we will choose the most suitable solution for our system. We will also describe some important parts that will become part of the future solution — for example, the data synchronization process.

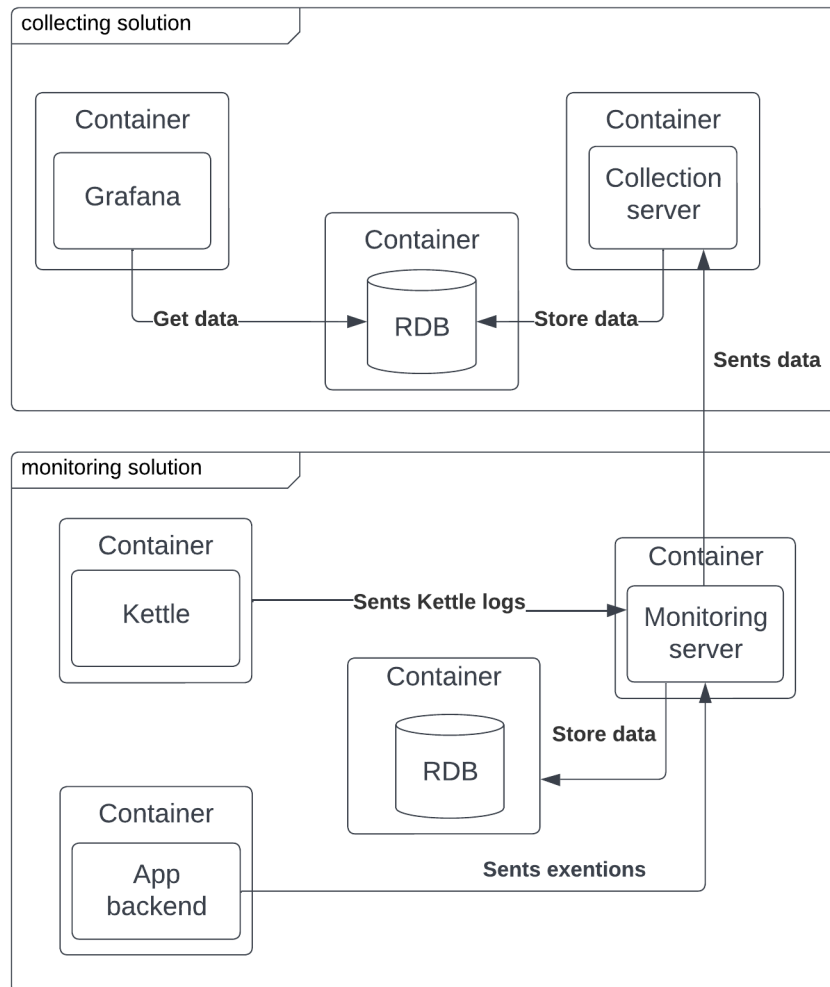
2.1 Solution design

This chapter will analyze two architectures based on different systems for collecting metrics. We are talking about solutions based on Prometheus and TICK stack. Let us first describe the common part of both systems before focusing on their differences. You can see the common part of the architecture in the following page.

As you can see, the solution consists of two independent components: monitoring and collection solutions. The monitoring solution resides in the hospital environment, while the collection solution is a separately running server that resides outside the hospital environment.

The monitoring solution sends data to the collection solution, which receives them. Communication, as we know from the requirements analysis, occurs only in one direction. In the client-server architecture, the monitoring solution becomes a client, and the collection solution becomes a server.

As you can see, most of the components are placed in containers. It means that the architecture uses a container engine. In our particular case, this is the Docker engine.



■ **Figure 2.1** The common architectures part

Monitoring solution

A key part of the monitoring solution is the monitoring server. It is responsible for sending data to the collection solution. In its turn, it receives data from two sources. These sources are the hospital application's backend and the Kettle, which sends information about job processing.

Information from these two sources is stored in a relational database before being sent to the server. Thus, the monitoring server is essentially responsible for synchronizing this database and the database that is part of the collection solution.

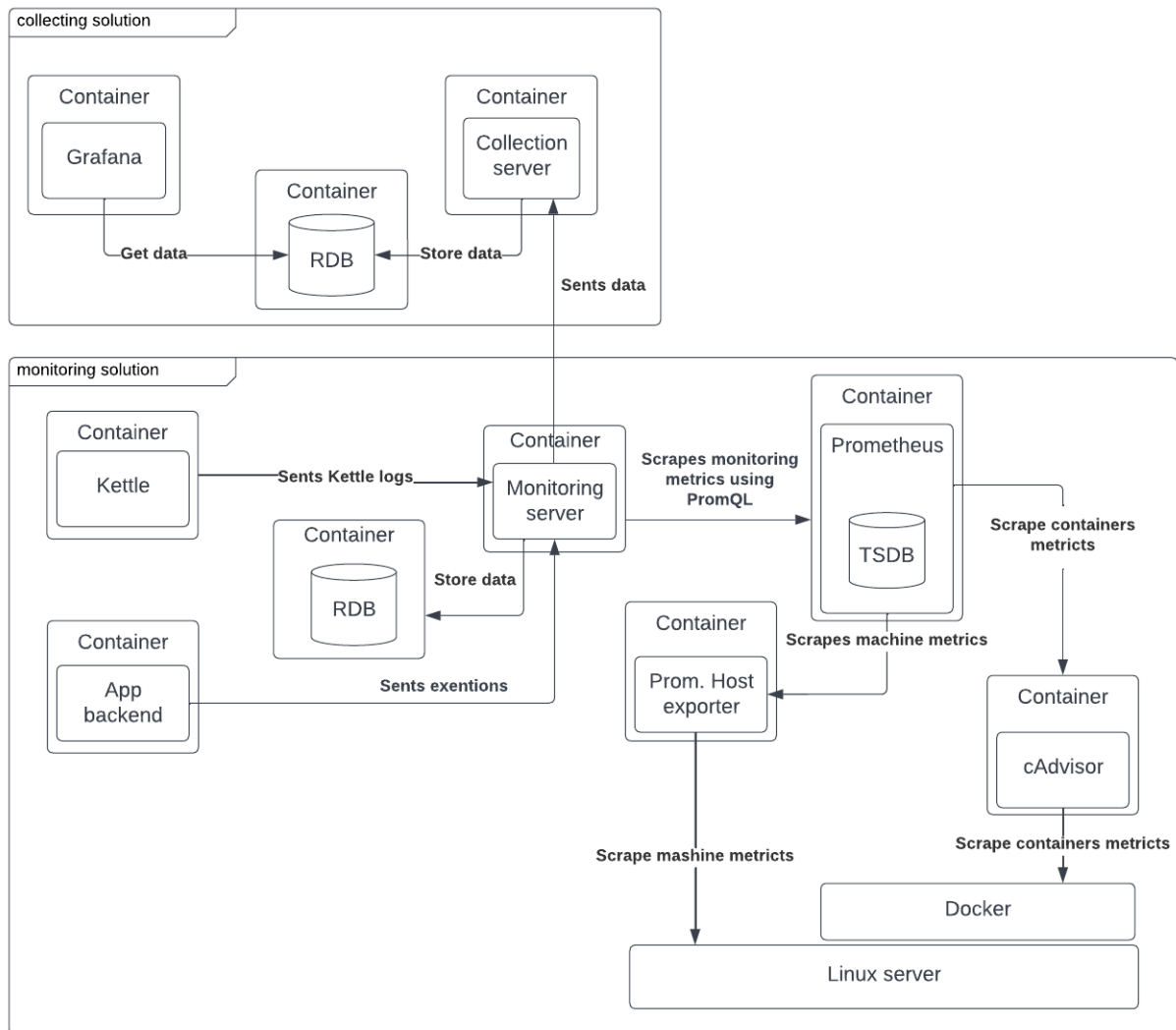
Collection solution

A key part of the collection solution is the collection server. It is responsible for data receiving and storing in the database. It is also responsible for database synchronization.

Another key part of our solution is Grafana. It is responsible for data and metrics visualization, which are stored in the database. It is also responsible for alerting. Thus Grafana solves failure prediction and visualization requirements.

2.1.1 Prometheus solution

Let us see what needs to be done to integrate Prometheus into our solution. First, let us illustrate this solution. You can see it at the following figure:



■ **Figure 2.2** Prometheus solution architecture

As you can see, the monitoring solution has three more components. The main component responsible for collecting metrics is Prometheus. Also, two components are used to receive metrics from the hospital server: Host exporter, which exports metrics directly from the Linux server, and cAdvisor, which collects metrics about Docker containers.

In this solution, we use the cAdvisor, since Docker natively supports Prometheus only in an experimental mode. It means that you need to run dockerd in experimental mode. [44]

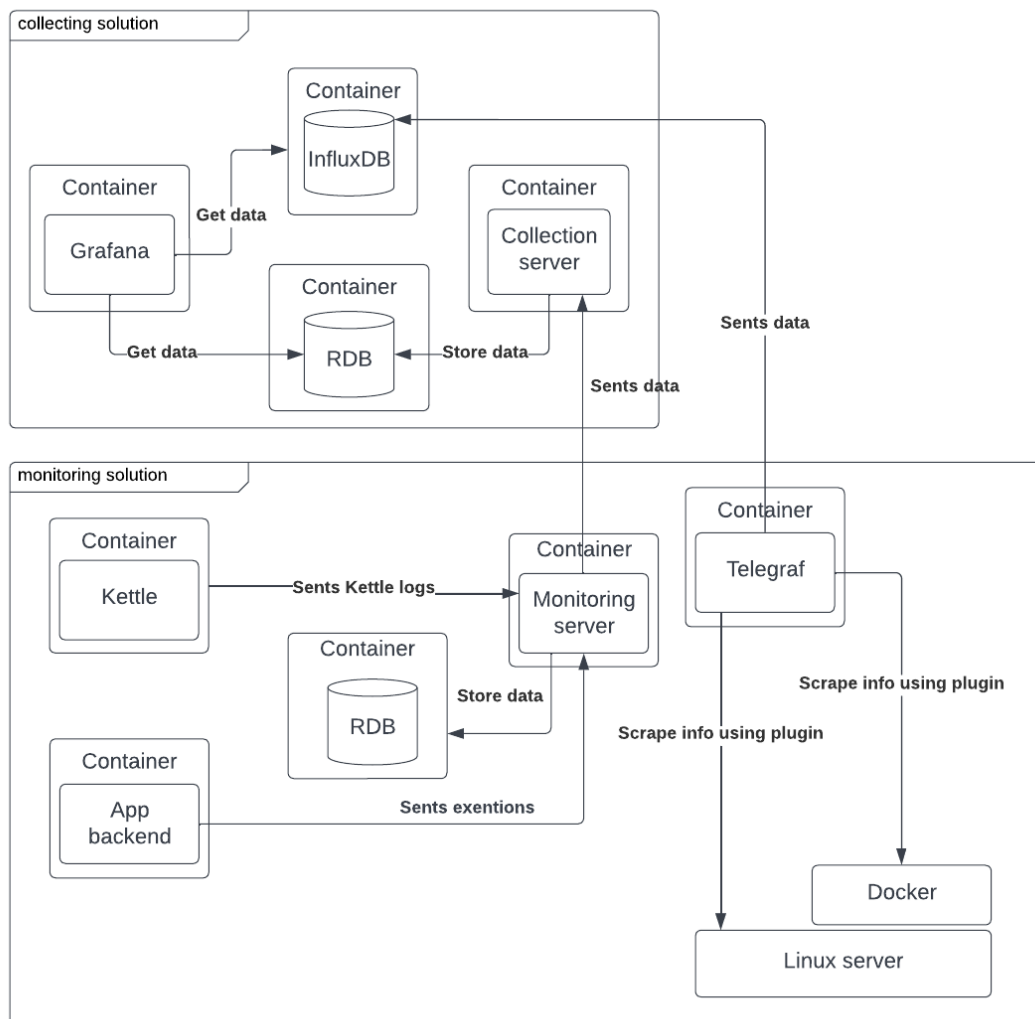
We don't want this in production, and cAdvisor is a popular solution for this problem.[45]

The collection of metrics works on the principle of scraping. The components provide APIs for collecting metrics. The Prometheus server scrapes them at certain intervals and then saves them to the time series database, which is part of the Prometheus server.

The Prometheus server also has an API, which the monitoring server uses to scrape metrics in certain intervals. Then the monitoring server sends it to the collection server, which stores it in a relational DB. At the end of this flow, Grafana visualizes the metrics from relational DB.

2.1.2 TICK stack solution

Let's see what needs to be done to integrate the TICK stack into our solution. First, let us illustrate this solution. Pay attention to the following figure:



■ **Figure 2.3** TICK stack solution architecture

This solution is pretty simple. There is one more component in the monitoring solution and one more in the collection solution. On the monitoring solution side, it is the Telegraf. It is responsible for collecting metrics, and then it sends these metrics to InfluxDB. InfluxDB stores this data, and finally, Grafana uses InfluxDB as a data source.

2.1.3 Solutions comparison

In this chapter, we will analyze the pros and cons of both solutions, and we will discuss their features as well as prospects for use.

Prometheus solution

In the case of Prometheus, the solution has some complexities related to our solution's requirements.

The first difficulty for us is one-way communication, and this requirement does not match the way Prometheus works. It would be beneficial to place the Prometheus server within the collection solution and send metrics by exporters to this server. However, Prometheus itself actively collects metrics and can be placed only as part of a monitoring solution.

This fact means that it is also necessary to ensure the transfer of metrics from the Prometheus server to the collection solution. This communication is built through a monitoring server and a collection server. And also, the monitoring server, in its turn, must scrape the metrics from the Prometheus server using PromQL.

Another difficulty is related to the Prometheus server database. This database is part of the Prometheus server and stores data in a specific format. This database is time series, not relational, and for metrics saving on the side of the collection server, it is necessary to transform them. That also adds complexity to the solution.

These two difficulties can be solved by Grafana Mimir. The Grafana Mimir is an open-source time series database, that supports metrics remote writes from Prometheus and metrics visualization in Grafana. [46]

It should be noted that although this component solves some problems, it makes the solution more resource-intensive and is a relatively new technology, the release of which was made a few months ago. Therefore, we do not propose the use of this technology in our solution.

TICK stack solution

In the case of the TICK stack, the one-way communication requirement is not a problem, and it is how it natively works. In the case of the TICK stack, storage is split with a part that scrapes metrics, and the scrape metrics part sends metrics directly to storage. There are no problems with integration.

Comparison

This section will discuss the pros and cons of using the two technologies, and we will also describe possible scenarios in which each technology could be used.

So let's see what aspects are better in case of the TICK stack usage:

Complex integration As mentioned earlier, Prometheus, in our particular case, has several difficulties. Due to these complexities, integrating Prometheus into our solution is more problematic than integrating a TICK stack.

Resource-intensive Prometheus is more expensive than the TICK stack in terms of CPU, memory, and others. The TICK stack is a much more suitable solution if we do not have a lot of resources on the monitoring solution side. Since we assume that there will be several instances of monitoring solutions, the resource savings will increase with the increase in the number of instances.

Scalability With the increase in the number of monitoring solution instances, several collection solutions will likely need to emerge. Also, a rather important issue, which has not yet been resolved in this work, is data replication. All this leads to the emergence of a cluster from the side of the collection solution. The TICK stack is designed in such a way that it allows using the Chronograph to synchronize data from several InfluxDB instances and is well-suited for creating a cluster. Prometheus, in its turn, does not offer such a use case.

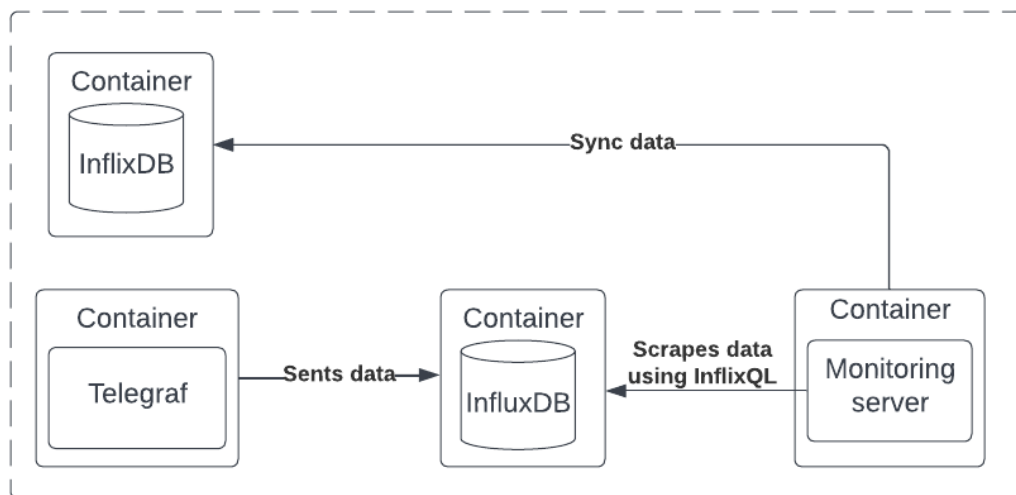
Now let's see what aspects are better in case of the Prometheus usage:

Metrics collection Prometheus should be used if we offer active growth on the side of the monitoring solution and add many external solutions to our solution. In such a case, using Prometheus is beneficial, as it offers a huge selection of exporters and client libraries. It provides a simpler collection of metrics compared to Telegraf, which cannot boast of such a large number of plug-ins for collecting metrics.

Saving data There is also another small advantage of the Prometheus solution. The metrics in this solution fall into persistence storage on both the monitoring solution and collection solution sides. While in the TICK stack solution, the metrics are saved only on the side of the collection solution. Because of this, a small problem arises if, for some reason, the collection server is not available for a long time.

Telegraf partially solves the problem and uses a buffer as temporary metrics storage. However, this buffer is stored in RAM and can easily overflow. Also, if the monitoring solution fails, the metrics collected when the collection server is unavailable will be lost.

Of course, this problem can be solved by adding a second instance of InfluxDB to the side of the monitoring solution and implementing synchronization between the two instances of InfluxDB through the monitoring server. This modification is shown in the following image:



■ **Figure 2.4** TICK stack solution modification

With this modification, the solution is still not as complicated as the one based on Prometheus. We currently assume that this solution may be useful in the future, but at the moment, it will not bring obvious benefits compared to the complexity of implementation.

2.1.4 Solutions design summary

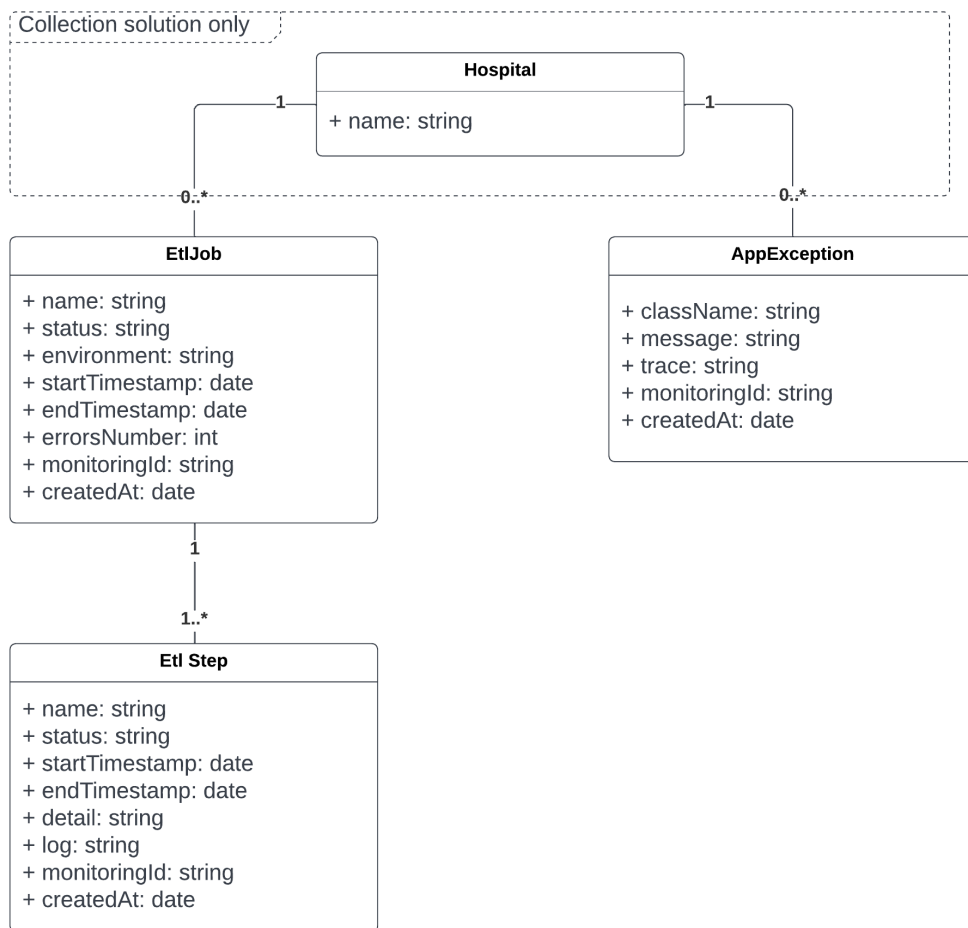
Each solution undoubtedly has its benefits and deserves to be implemented. However, the most likely scenario would be the need for clustering on the part of the collection solution.

We should also consider that the monitoring solution is currently a small set of components, and the resource use of Prometheus is comparable with the resource usage of the components we will monitor. In the case of the Telegraf, resource usage is much less, and the use of the Telegraf looks more rational and reasonable.

Thus, the monitoring solution used in the TICK stack is currently the most fortunate. In this regard, it will be implemented in the future.

2.2 Domain model

A rather important design chapter is creating a data model synchronizing the monitoring and collection solutions. This model represents the data that will be involved in synchronization between the collection and monitoring servers. Let us get acquainted with the domain model, which is shown in the following figure:



■ Figure 2.5 Domain model

As we can see, we are getting two independent collected data from two components: the Kettle and the hospital application backend. It is important to note that since we are synchronizing data between monitoring and collection solutions, the domain data model is identical. You can also see that a hospital entity is added from the side of the collection solution, which will identify the data for each hospital.

2.3 Synchronization design

This chapter will design synchronization between collection and monitoring servers as required. Based on the requirements analysis, there are five main requirements, and they are as follows:

- The monitoring service will be able to send data to the collection server periodically in batches.
- The monitoring service will be able to collect data from the hospital environment.
- Data consistency. The data must be consistent between monitoring and collection services over time, meaning that data should not be lost or duplicated.
- Failure requirement. After the monitoring service or collection service fails, the data in databases must be synchronized correctly.
- Communication must be one-directional. It means that the monitoring solution can request a collection solution but not vice versa.

It is important to say that we do not intend to delete or transform the data. This solution assumes only the collection of statistics and its visualization on the side of the collection solution. In other words, our task is about insert synchronization.

Let's look at the problems that need to be solved for correct synchronization:

- It is necessary to send in the new batch that has not yet been sent.
- It is necessary to know if the state of the collection server's database matches the state of the monitoring server's database at the moment before sending a new batch. In other words, the party was saved successfully.
- In case of an unsuccessful save, it is necessary to know the state of the collection server for the new batch to be created correctly.

We will achieve successful insertion synchronization by solving these problems and keeping one-way communication.

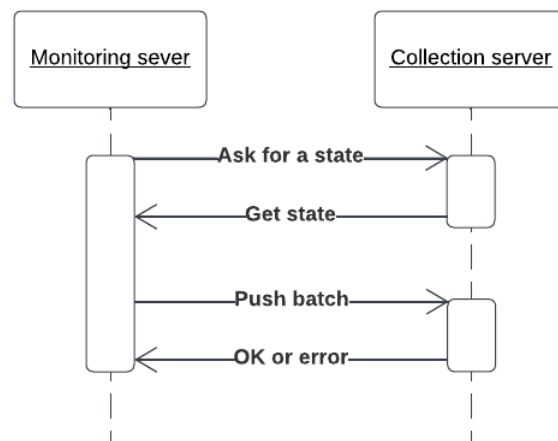
2.3.1 Naive synchronization solution

First, we will describe the naive solution, and then we will propose an improvement that will reduce the number of requests.

Let us define the characteristics of a naive solution:

- In the naive solution, synchronization occurs as follows.
- We never know the state of the database on the side of the collection server, and before creating a batch, we ask the collection server for this state.
- For us, the success of the write operation on the collection server is unimportant. Since we always ask about its status before sending a new batch.

This process is shown in the sequence diagram:

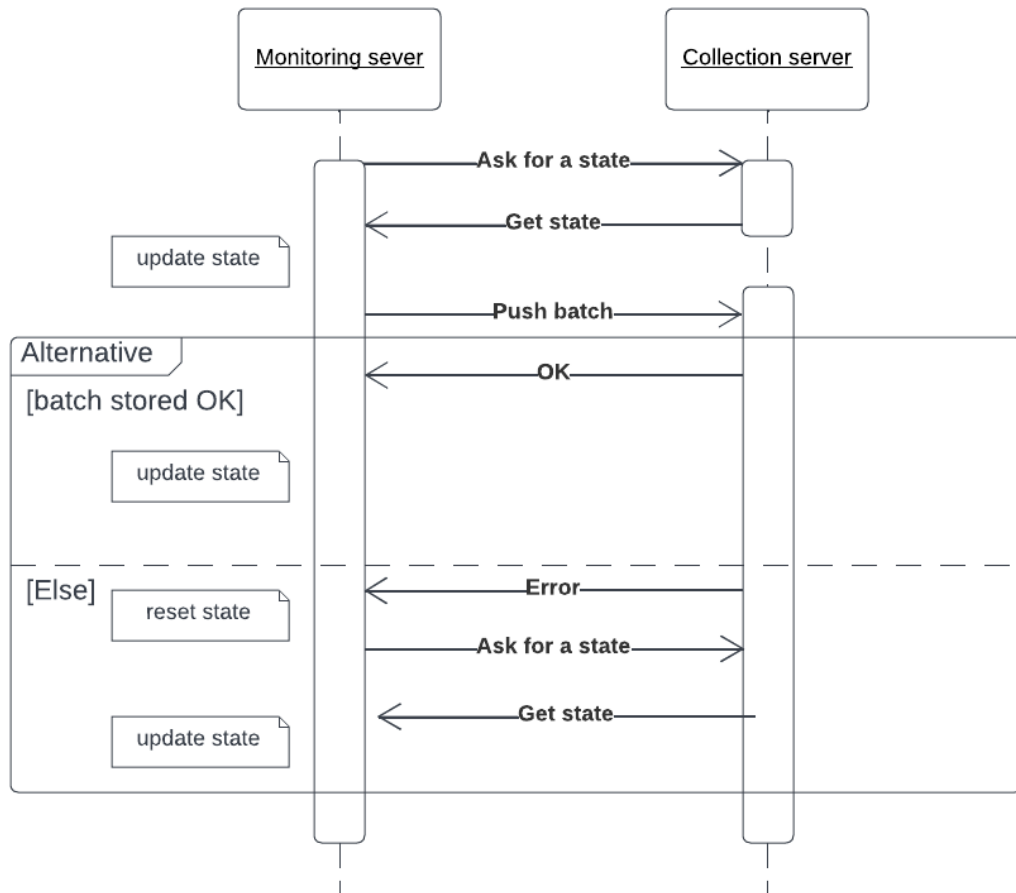


■ **Figure 2.6** Naive synchronization solution

As part of our solution, we propose generating a key on the monitoring server side for each entity we want to send to the collection server. Then the state will be the key of the last record, which is stored on the side of the collection server. Knowing this on the monitoring server side will simplify batch creation. To take all the instances that are newer than the one whose key is the state is enough to do this.

2.3.2 Remember state synchronization solution

The following solution involves keeping the state in RAM on the side of the monitoring server. This solution is based on the fact that we can calculate the state we will get from the collection server if the request succeeds. Therefore, we will update the status for each batch that we sent successfully. If the batch was not sent successfully or there were other problematic situations, the status information is deleted, and we have to update the status. In other words, ask in what state the collection server is. This process is shown in the sequence diagram:



■ **Figure 2.7** Remember state synchronization solution

2.3.3 Synchronization design summary

We have designed how batches should be synchronized. We have proposed two solutions, each of them meets the requirements and can be used for the solution. It is more reasonable to use an improved communication version, which is the last one, so this version will be implemented.

2.4 Design summary

In this chapter, we have designed the architecture of several metrics collection solutions and selected the one that suits us best. We also figured out what data we would have to synchronize and designed synchronization that would allow us to transfer data correctly.

Thus, after analyzing the requirements for the system and designing its most difficult moments, we are ready to start implementation.

Implementation

In this chapter, we will describe how the final solution was implemented. We will describe in detail the work and interaction processes for the individual components of the two systems: monitoring and collection solutions. We will also describe the process of communication between these two solutions.

In addition, we will analyze in detail how we implemented such requirements as synchronization, security, data visualization, and failure prediction.

3.1 Monitoring solution infrastructure

This section will describe how the monitoring solution infrastructure is implemented, what components it consists of, and what processes take place within it.

3.1.1 Introduction

The monitoring solution consists of several components. These components are:

- Monitoring server
- Telegraf instance
- Monitoring database
- Application backend
- Application frontend
- Application database
- Kettle instance

The last four components have already been implemented as part of the monitoring solution, so we will not describe their work in detail. Instead, we will focus on describing the components that we have implemented.

It is important to note that several techniques have been applied to simplify the work with these components. First of all, containerization was carried out.

Containerization provides a component with operating system libraries and the necessary dependencies to create a container – an executable file that works stable in any infrastructure. Since Docker was chosen as the container engine, we can call the process dockerization.

Secondly, since our solution contains many components, there is a solution to facilitate the configuration, management, and running of many Docker containers – Docker Compose.

“Docker Compose is a tool that was developed to help define and share multi-container applications. With Compose, we can create a YAML file to define the services and with a single command, can spin everything up or tear it all down.” [47]

Thanks to these technologies, creating an infrastructure consisting of several components is quite simple. We bring to your attention the Docker Compose configuration file, which contains the information necessary to create these components:

```

1  version: "3.6"
2  services:
3    telegraf:
4      container_name: monitoring-telegraf
5      image: telegraf:1.18-alpine
6      volumes:
7        - ./telegraf/telegraf.conf:/etc/telegraf/telegraf.conf:ro
8        - /var/run/docker.sock:/var/run/docker.sock
9      ports:
10     - ${TELEGRAF_PORTS}
11     user: root
12     entrypoint: "telegraf"
13     env_file: configuration.env
14
15     monitoring_server:
16       depends_on:
17         - monitoring_db
18       container_name: monitoring-server
19       image: ${MONITORING_IMAGE}
20       ports:
21         - ${MONITORING_PORTS}
22       env_file: configuration.env
23
24     monitoring_db:
25       image: postgres:13.1-alpine
26       container_name: monitoring-db
27       ports:
28         - ${DB_PORTS}
29       env_file: configuration.env
30       volumes:
31         - ./data/db:/var/lib/postgresql

```

■ **Listing 3.1** Docker Compose file for monitoring solution

Let us take a closer look at the features described in this configuration file. As you can see, there are three components implemented on the monitoring solution side, and each of them corresponds to the container.

The first of them is the Telegraf instance. For this component, we used the existing image `telegraf:1.18-alpine` and configured it using the configuration file `telegraf.conf`, which you can see as one of the volumes. As another volume, you can see the `docker.sock`, which we are using, as we knew from the analysis part, to access the docker metrics. Using the volumes, we are specifying mount points that we share with the container. In other words, the container can access these mount points.

The second one is the monitoring server. It uses the image that we create using the jib plugin. The name of the target image is defined in `${MONITORING_IMAGE}` variable.

The last one is the monitoring database. Same as for the Telegraf, there is also used an already existing image. Now, this is an image of the PostgreSQL database.

It needs to mention that all of these components have added an environment file that calls `configuration.env`. It contains the variables used to configure each of the containers. We will provide more information about it in the description of the components.

3.1.2 Monitoring server

This component implements the data synchronization process. The monitoring server provides an API that is used to obtain this data by the Application backend and Kettle instance. Then, the monitoring server sends this data at certain intervals in batches.

This component is an application implemented using Spring and Gradle and written in the Kotlin language. Detailed information about implementation will be provided in the sections below. This section will focus on how this component is integrated into the infrastructure.

To the infrastructure, it was integrated as a ready-made image. The image is assembled using the jib plugin. This plugin is added to `build.gradle.kts` as a configuration.

```
1 jib {
2     to {
3         image = "monitoring-server"
4         tags = setOf("latest")
5     }
6 }
```

■ Listing 3.2 jib plugin usage

For now, it looks simple, but as a future improvement, it can be supplemented with versions and settings for the Docker Hub.

The other interesting part of monitoring server infrastructure is configuration. A standard Spring application configuration is solved as `application.properties` files. In our case, to define variables on the container level, we need to add them as environment variables to the Docker Compose configuration. Environment variables can be added to a separate file to make your Docker Compose configuration more readable:

```
1 SERVER_PORT=8081
2 COLLECTION_API_URL=host.docker.internal:4003/api
3 HOSPITAL_NAME=bestHospitalEver
4 DB=root
5 DB_USERNAME=root
6 DB_PASSWORD=root
7 DB_CONTAINER_NAME=monitoring-db
8 DB_PORT=5432
9 # cron rates in ms, default (10s)
10 APP_EXCEPTION_CRON_RATE=10000
11 ETL_CRON_RATE=100000
```

■ Listing 3.3 configuration.env for monitoring server

There are environment variables that are configured for monitoring the server. The same variables are propagated to `application.properties` file:

```

1 server.port=${SERVER_PORT}
2 collection.api.uri=${COLLECTION_API_URL}
3 hospital.name=${HOSPITAL_NAME}
4 # DataSource
5 spring.datasource.url=jdbc:postgresql://${DB_CONTAINER_NAME}:${DB_PORT}/${DB}
6 spring.datasource.username=${DB_USERNAME}
7 spring.datasource.password=${DB_PASSWORD}
8 spring.datasource.driverClassName=org.postgresql.Driver
9
10 spring.jpa.hibernate.ddl-auto=update
11
12 cron.rate.app-exception=${APP_EXCEPTION_CRON_RATE}
13 cron.rate.etl=${ETL_CRON_RATE}

```

■ **Listing 3.4** `application.properties` file with environment variables

In this way, environment variables set configuration variables. It allows you to configure the container from the outside. It also allows you to define variables in one place and avoid configuration mistakes.

In the following chapters, we will also get acquainted with the configuration related to specific areas: security, synchronization, and others.

3.1.3 Telegraf instance

This component implements the collection of metrics and sends them to InfluxDB, which is located on the side of the collection server. In its turn, Telegraf also implements synchronization of metrics, but we will analyze this in the Synchronization section.

Since the Telegraf is an external component, we can only configure it. The main configuration file is `telegraf.conf`. It contains Telegraf's general settings and also input and output plugins. In the next figure, you can see the Telegraf general settings:

```

1 # Global tags can be specified here in key="value" format.
2 [global_tags]
3   hospital = "$HOSPITAL_NAME"
4 # Configuration for telegraf agent
5 [agent]
6   # Default data collection interval for all inputs
7   interval = "$DEFAULT_DATA_COLLECTION_INTERVAL"
8   # flush this buffer on a successful write.
9   metric_buffer_limit = $BUFFER_SIZE
10  ...
11  collection_jitter = "0s"
12  flush_interval = "1s"
13  flush_jitter = "0s"

```

■ **Listing 3.5** The general Telegraf settings

In the general settings of the Telegraf instance, you can set up an interval in that data will be extracted by input plugins and sent to output plugins. It also allows you to define the buffer that will store the data on the monitoring solution side when the collection solution is disabled.

In the global configuration, we also define some global tags that will be added to each collected measurement. For us, it is the unique hospital identifier, and for that, we will group data collected from each hospital.

Next, an important part of setting up Telegraf is connecting plugins. Let us get acquainted with input plugins used to receive the data from the host OS and the Docker infrastructure. In the next listing, you can see the configuration of input plugins used in our solution:

```
1  # Read metrics about docker containers
2  [[inputs.docker]]
3  # Docker Endpoint
4  endpoint = "$DOCKER_MONITORING_ENDPOINT"
5  perdevice = true
6  total = false
7  docker_label_include = []
8  docker_label_exclude = []
9  # Which environment variables should we use as a tag
10 tag_env = ["JAVA_HOME", "HEAP_SIZE"]
11
12 # Input plugins for collecting metrics from system
13 [[inputs.cpu]]
14   percpu = true
15   totalcpu = true
16   fielddrop = ["time_*"]
17
18 [[inputs.disk]]
19   ignore_fs = ["tmpfs", "devtmpfs"]
20 [[inputs.diskio]]
21 [[inputs.kernel]]
22 [[inputs.mem]]
23 [[inputs.processes]]
24 [[inputs.swap]]
25 [[inputs.system]]
26 [[inputs.net]]
27 [[inputs.netstat]]
28 [[inputs.interrupts]]
29 [[inputs.linux_sysctl_fs]]
```

■ **Listing 3.6** The input plugins Telegraf settings

In this configuration file, each instance of `[[inputs.*]]` represents the separate plugin for metrics collection. You can see here the Docker input plugin has defined an endpoint setup, depending on the OS. Also, you can see several host OS inputs plugins, that are different for each metric group (disk, CPU, memory, and others).

There are also output plugins that we have in the Telegraf configuration file. In the next listing, you can see the configuration of output plugins used in this solution:

```

1  # Configuration for influxdb server to send metrics to
2  [[outputs.influxdb]]
3    urls = ["$INFLUX_URL"] # required
4
5    # Write timeout (for the InfluxDB client), formatted as a string.
6    timeout = "5s"
7
8    username = "test"
9    password = "test"

```

■ **Listing 3.7** The output plugins Telegraf settings

There is only one plugin in the output configuration. It is a plugin for InfluxDB that setups the communication with the InfluxDB instance on the collection solution side. It also has some important settings as the URL of the InfluxDB instance and intervals, in which we will send the data to the InfluxDB.

3.1.4 Monitoring database

This component saves the data received by the monitoring server. Thus, this component participates in synchronization since synchronization is carried out between two databases — one on the side of the collection solution and this one.

This component has no extra configuration file. All configuration was realized by environment variables such as ports, user data, database, and others.

3.2 Collection solution infrastructure

This section will describe how the collection solution infrastructure is implemented, what components it consists of, and what processes take place within it.

3.2.1 Introduction

The collection solution consists of several components. These components are:

- Collection server
- InfluxDB instance
- Collection database
- Grafana instance

As in the case of monitoring solutions, containerization was applied to simplify working with these components, and Docker Compose was used.

Let us bring to your attention the Docker Compose configuration file, which contains the information necessary to create these components:

```
1 version: '3.6'
2 services:
3   influxdb:
4     image: influxdb:1.8-alpine
5     container_name: collection-influxdb
6     env_file: configuration.env
7     ports:
8       - ${INFLUX_PORTS}
9     volumes:
10      - ./influxdb/imports:/imports
11      - ./influxdb/conf/influxdb.conf:/etc/influxdb/influxdb.conf
12      - influxdb_data:/var/lib/influxdb
13
14   grafana:
15     image: grafana/grafana:9.0.0
16     container_name: collection-grafana
17     restart: always
18     depends_on:
19       - influxdb
20     env_file: configuration.env
21     links:
22       - influxdb
23     ports:
24       - ${GRAFANA_PORTS}
25     volumes:
26      - grafana_data:/var/lib/grafana
27      - ./grafana/provisioning:/etc/grafana/provisioning/
28      - ./grafana/dashboards:/var/lib/grafana/dashboards/
29      - ./grafana/conf/grafana.ini:/etc/grafana/grafana.ini
30
31     # simmular as monitoring_server
32     collection_server:
33     # simmular as monitoring_db
34     collection_db:
35
36 volumes:
37   grafana_data: {}
38   influxdb_data: {}
```

■ Listing 3.8 Docker Compose file for collection solution

Let us take a closer look at the features described in this configuration file. As you can see, there are three components realized on the collection solution side, and each of them corresponds to the container.

The first of them is the InfluxDB instance. For this component, we used an already existing image `influxdb:1.18-alpine` and configured it using the configuration file `influxdb.conf`, which you can see as one of the volumes. As another important volume, you can see the `influxdb_data`, which is added as the volume at the bottom of the configuration file. It creates the volume and saves the InfluxDB data, settings, and other information to this volume.

The second one is the Grafana instance. As with the InfluxDB, we use the existing image `grafana:9.0.0`. It uses its own internal database to store the data. We create an `grafana_data`

volume, the same as for the InfluxDB. Then, we also create volumes for configurations that we need to share. It includes dashboards and data source configurations that we use for visualization.

There are also two services whose configurations are similar to the configuration of related services at the monitoring solution part. These services are the collection server, which is configured similarly to the monitoring server, and the collection database, that configured the same as the monitoring one.

3.2.2 Collection server

This component participates in the data synchronization process. In order to get this data, the server collection provides an API that several monitoring solutions use server monitoring to pass data from monitoring databases. The collection server stores this data in the collection database, thus completing the synchronization.

Like the monitoring server, this component is an application implemented using Spring and Gradle and written in the Kotlin language.

The collection server is configured similarly to the monitoring server. It also uses jip plugin to create an image and configure it the same way how monitoring server did.

3.2.3 Collection database

This component saves the data received by the collection server. Thus, it participates in synchronization. The synchronization is carried out between two databases — one on the side of the monitoring solution and this one. In the future, these data will be used by Grafana to visualize and predict failures in the monitoring solutions.

The collection database has an identical configuration as the monitoring one. It also uses the PostgreSQL image with the same environment variables.

3.2.4 InfluxDB instance

This component saves the metrics received from the Telegraf instances and keeps them on the side of the collection solution. These metrics are used by Grafana to visualize and predict failures of monitoring solutions.

The configuration of the InfluxDB instance is placed at the `influxdb.conf` file, that, as you can see in the Docker Compose configuration, is added as a volume. Let us bring to your attention the configuration file, which contains the information necessary to set up the InfluxDB instance:

```
1 [meta]
2   dir = "/var/lib/influxdb/meta"
3
4 [data]
5   dir = "/var/lib/influxdb/data"
6   engine = "tsm1"
7   wal-dir = "/var/lib/influxdb/wal"
8
9 [http]
10  enabled = true
11  bind-address = ":8086"
12  auth-enabled = true
13  log-enabled = true
14  write-tracing = false
15  pprof-enabled = true
16  pprof-auth-enabled = true
17  debug-pprof-enabled = false
18  ping-auth-enabled = true
19  https-enabled = true
20  https-certificate = "./influxdb.pem"
21  shared-secret = "my super secret pass phrase"
```

■ **Listing 3.9** The InfluxDB instance configuration

There are three parts to the InfluxDB configuration. The first two parts configure data and metadata storage. The last configures the HTTP communication details, such as authentication, HTTPS parameters, and others. The other part of the configuration, such as user credentials, is defined as environment variables.

3.2.5 Grafana instance

Grafana performs data visualization, and failure prediction, and sends alerts in our solution.

It has a very extensive configuration. Grafana is able to configure the dashboards for your metrics visualization, create alert rules depending on these metrics, and configure data sources from which it will get the metrics. There are some related things that also can be configured, such as folders, permissions, contact points, notification policies, and many others.

In our Grafana configuration, we will focus on the configuration of dashboards, data sources, alert rules, and related folders. However, it can be extended in the future for more complicated configuration.

All these things you can easily configure using the Grafana UI. But the main problem with the configuration is that you will have to export this configuration, or you need to reconfigure it each time when you need to launch the new Grafana instance.

Grafana supports dashboard exporting, and you can export them as JSON files. The export or configuration of other things is not yet well thought out by Grafana and has become a deep pain of our solution.

Fortunately, Grafana offers not only configuration using UI. You can configure it manually using the YAML format or configure needed entities via Grafana API [48].

First of all, we tried to configure data sources and alerts using the YAML configuration. It didn't take long to create the data sources configuration. This configuration is too long to add to this work. Instead of it, we will describe the one data source configuration. In the next figure, you can see the InfluxDB data source configuration:

```
1  datasources:
2    - name: InfluxDB
3      type: influxdb
4      access: proxy
5      orgId: 1
6      uid: influxdb_uid
7      url: http://collection-influxdb:8086
8      user: "test"
9      database: "influx"
10     withCredentials:
11       isDefault: true
12     jsonData:
13       timeInterval: "5s"
14       graphiteVersion: '1.1'
15       tlsAuth: true
16       tlsAuthWithCACert: true
17     secureJsonData:
18       password: "test"
19     version: 1
20     editable: true
```

■ **Listing 3.10** The InfluxDB data source configuration for the Grafana

The configuration contains the necessary settings for the data source, such as URL, the time interval for data receiving, security settings, and other information about the data source.

The data source configuration takes up no more than a hundred lines and looks quite compact. Despite this, the configuration took a relatively long time. It is worth noting that the alert rules configuration would take up several thousand lines. Of course, this is a very bad option to configure alerting manually.

A good solution was to combine the Grafana UI and the Grafana API ways. It is a relatively simple task to define an alert rule using Grafana UI and get it using Grafana API GET method for `/API/v1/provisioning/alert-rules/{UID}` endpoint. The received in JSON format alert rule configuration can be used with a little modification to create this rule another time via the Grafana API. It can be simply realized by sending this configuration as POST request to the same endpoint.

This export was realized as the Postman collection and can be easily modified and reused. This configuration also takes up several thousand lines, but now they are generated by the Grafana UI.

In the following sections, we will take a closer look at the functionality that Grafana offers.

3.2.6 Conclusion

This section described the implementation of the collection solution infrastructure in detail. We also examined the technologies used both for the whole solution and for individual components implementation.

Next, we will study how the individual requirements for the complete solution were met and also describe in more detail the individual parts of the components described earlier.

3.3 Synchronization

In this section, we will focus on how synchronization was implemented. In our solution, both metrics and data are transferred. Both are synchronized. Let us provide the information about each of these cases separately.

3.3.1 Data synchronization

Data synchronization in our solution occurs between monitoring and collection databases. Synchronization, as such, is realized by monitoring and collection servers.

In the design chapter, we chose stateful synchronization, so we suggest you familiarize yourself with how it was implemented. The state is pictured in the following listing:

```
1 data class State (  
2     val lastSendMonitoringId: String = "",  
3     val stateName : StateName = StateName.IN_PROCESS  
4 ){}
```

■ **Listing 3.11** The synchronization state implementation

As you can see, it is represented by two values: the state and the `lastSendMonitoringId` (LSMI) value, which identifies the last synchronized record. In other words, LSMI is the last instance of data, that was successfully transferred to the monitoring server and stored in the collection database. The value of LSMI is non-zero only for the `IN_PROCESS` state. The other two states have this value equal to null.

The synchronization process goes through some states, and there are only three of them:

- `NOT_INIT` – the state corresponding to the absence of communication between the monitoring server and the collection server.
- `START` – the state in which the synchronization process starts.
- `IN_PROCESS` – the state in which synchronization occurs or resumes after the communication is interrupted.

For now, let us focus your attention on the synchronization process. The process starts in the `NOT_INIT` state. At the moment, it is unknown what state the collection server is in. To do this, we send a request to set the state of the collection server. Then, depending on whether there are saved data records on the side of the collection server or not, the monitoring server goes either to the `IN_PROCESS` state or to the `START` state.

If we are in the `START` state, we send the first sync request, and then, upon receiving a successful response, we move to the `IN_PROCESS` state and remember the LSMI value for the last record, that we received.

The synchronization request is then repeated after a certain amount of time, and we remain in the `IN_PROCESS` state and update the LSMI if the request is successful. Otherwise, we enter the `NOT_INIT` state.

From the `NOT_INIT` state, we send a request to receive LSMI, and if the request is successful and LSMI is, it switches to the `IN_PROCESS` state. Otherwise, we remain in the `NOT_INIT` state.

The following code demonstrates the implementation of this process from the monitoring server side:

```

1 fun syncTask() {
2     try {
3         if (syncBusiness.isStateNotInit()){
4             syncBusiness.updateState(
5                 communicationBusiness.pullState()
6             )
7         }
8         val batch = syncBusiness.createBatch()
9         val state = communicationBusiness.pushBatch(batch)
10        syncBusiness.updateState(state)
11    } catch (e: Exception){
12        // state is back to NOT_INIT
13    }
14 }
15 ...
16 // createBatch in syncBusiness
17 fun createBatch(): List<Synchronizable> {
18     if (isStateStart()){
19         val batch = getBatchAtStart()
20         if (batch.isNotEmpty()){
21             updateState(State(batch.last().monitoringId))
22             return batch
23         }
24     }
25     if (isStateInProgress()){
26         val batch = getBatchInProgress()
27         if (batch.isNotEmpty()){
28             updateState(State(batch.last().monitoringId))
29             return batch
30         }
31     }
32 }

```

■ **Listing 3.12** The synchronization from monitoring server side

The `syncTask()` runs by the Cron for each synchronized data (ETL jobs and Application exceptions). At the beginning of the communication, it asks the collection server for a state that is the collection database. The implemented logic on the collection server side is next:

```

1 @PostMapping("/exception/state")
2 fun getAppExceptionState(@RequestBody req: StateRequest): State {
3     return try {
4         val monitoringId = syncRepository.getLastMonitoringId(req.hospitalName)
5         State(monitoringId, StateName.IN_PROCESS)
6     } catch (e: Exception){
7         State(stateName = StateName.START)
8     }
9 }

```

■ **Listing 3.13** The synchronization from collection server side

On the collection server side, it has a simple realization. In case synchronization is interrupted, we find the last monitoring id and send it to the monitoring server with `IN_PROCESS` state. In case of synchronization starts, we send only a `START` state.

Next, let us bring to your attention the synchronization of metrics.

3.3.2 Metrics synchronization

The synchronization of metrics occurs between the Telegraf, on the side of the monitoring solution, and the InfluxDB, on the collection solution side.

However, this synchronization, unlike data synchronization, does not always occur. Data from Telegraf to InfluxDB are sent in intervals configured in `telegraf.conf`.

Synchronization becomes after the collection solution failure. If the collection solution, for some reason, does not process incoming requests, Telegraf caches the metrics into a special buffer, the size of which is configured in the Telegraf configuration.

As soon as the collection solution becomes available, Telegraf performs the data synchronization process. In this case, synchronization occurs between the Telegraf buffer and the InfluxDB instance. Thus, data are not lost if the collection solution fails, provided that a sufficient buffer value is configured.

3.4 Security

This section will focus on how security has been implemented within our solution. In our solution, both metrics and data are transferred. Let us mention that, the metrics transfer is realized between the Telegraf and the InfluxDB, and the data transfer is realized between the monitoring and the collection solutions. Security has been implemented for both options. Let's look at each case separately.

3.4.1 Data security

This section will look at how secure communication between collection and monitoring servers has been implemented.

Since our solution assumes certificate-based authentication, let us provide the information about what we need to implement this authentication method.

We use certificate authentication on both the server side and the client side. It allows the client and server to identify each other and create a secure connection uniquely. To implement this authentication, you need to have three pairs of certificates:

- root certificates – They are required for signing in our server-side and client-side certificates.
- server-side certificates – They are needed for authentication server by a client.
- client-side certificates – They are needed for authentication client by a server.

The creation of server-side certificates and client-side certificates proceeds similarly. In the beginning, a pair of keys is generated: public `some.key` and private `some.csr`. The same pair of root certificates then sign each pair of certificates. After that, the certificates are ready for use.

Also, since we are using Spring, it is good practice to store keys with the KeyStore. So we add client-side certificates to the KeyStore repository on the client side. Moreover, do the same thing with server-side certificates on the server side. Next, we also need to add root certificates to the server-side storage so the server can trust certificates signed with root certificates.

For both the client and the server certificates in the KeyStore, we store certificates in pairs, so as a result of transformations, we get the file `some.jks`. The last necessary step is to add the certificates to the configuration files.

Next, let us explore configuration examples for collection and monitoring servers:

```

1  ### collection server security configuration ###
2  # server CA
3  server.ssl.trust-store=classpath:ssl/truststore.jks
4  server.ssl.trust-store-password=testOnly
5  server.ssl.client-auth=need
6  # keystore need to be explicitly declared for Tomcat
7  server.ssl.key-store=classpath:ssl/keystore.jks
8  server.ssl.key-store-password=testOnly
9  ...
10 ### monitoring server security configuration ###
11 server-store=classpath:ssl/clientCA/keystore.jks
12 server-store-password=testOnly

```

■ **Listing 3.14** Collection and monitoring servers security configuration

As you can see, on the monitoring server side, there is only a client certificate configuration that is used for authentication of the monitoring server as a client. At the collection server, there is a corresponding part of configuration client-side authentication via a certificate. There is also a server certificate authentication configuration that is also used for HTTPS communication.

Moreover, it is necessary to implement authorization and authentication to the security on the side of the collection server.

To do it, you need to solve authorization and authentication for the collection server.

The authorization is pretty simple to implement. You just need to add `PreAuthorize` annotation before each endpoint that needs to be authorized.

```

1  @RestController
2  @RequestMapping("/api/app")
3  @PreAuthorize("hasAuthority('ROLE_USER')")
4  class AppController(@Autowired val appService: AppService){
5  ...
6  }

```

■ **Listing 3.15** Collection server authorization

For now, there is only the user with role `ROLE_USER` can access the `/api/app` endpoint. Next, we will focus on the authentication of this user.

To implement the authentication via client certificates in Spring Security, we must override the `SecurityFilterChain` bean. Below you can see an example of implementing authentication using a client certificate:

```

1  @Configuration
2  @Profile("security")
3  @EnableGlobalMethodSecurity(
4      prePostEnabled = true, securedEnabled = true, jsr250Enabled = true
5  )
6  class SecurityConfiguration() {
7
8      @Bean
9      @Throws(java.lang.Exception::class)
10     fun filterChain(http: HttpSecurity): SecurityFilterChain? {
11         http.authorizeRequests().anyRequest().authenticated()
12             .and().x509().subjectPrincipalRegex("CN=(.*?)(?:,|$)")
13             .userDetailsService(userDetailsService())
14         http.cors().and().csrf().disable();
15         return http.build()
16     }
17
18     @Bean
19     fun userDetailsService(): UserDetailsService? {
20         return UserDetailsService { username ->
21             if (username == "Alfa") {
22                 return@UserDetailsService User(
23                     username, "",
24                     AuthorityUtils
25                         .commaSeparatedStringToAuthorityList("ROLE_USER")
26                 )
27             }
28             throw UsernameNotFoundException("User not found!")
29         }
30     }
31 }

```

■ **Listing 3.16** Collection server security configuration

This certificate authentication example is for demonstration purposes only. For actual production use, the `UserDetailsService` should look slightly different and be based on the actual certificate settings.

However, it is also a correct authentication certificate, signed by the root certificate, but some certificate details are missing. For this authentication example, the valid certificate should have a username equal to Alfa.

In other words, the client authenticated by the client certificate, signed by the root certificate, and that will have username equals Alfa will be authorized as `ROLE_USER`.

In this section, we got acquainted with how collected data are protected. Next, we will focus on the metrics security.

3.4.2 Metrics security

This section will examine how secure communication between the Telegraf and InfluxDB servers has been implemented.

Unfortunately for us, InfluxDB did not support authentication using client certificates, and the highest security option that InfluxDB can offer is JSON Web Token usage. For the certificate

on the server side, this option is, of course, supportable, which allows us to create a secure HTTPS connection.

A secure communication setting requires configuration on both sides of the communication.

On the Telegraf side, the configuration is simple. You must add the correct credentials, such as database, username, and password. It should correspond with the InfluxDB environment variables. You should not also forget to switch the `http://` to `https://` in the URL's definition. You can see the required parameters in the figure below:

```

1  urls = ["$INFLUX_URL"]
2  database = "influx"
3  ...
4  username = "test"
5  password = "test"

```

■ **Listing 3.17** The Telegraf side security configuration

On the InfluxDB side, you must set up many parameters to make security work.

For the authentication, you need to enable it and configure some related details, such as the `ping-auth-enabled`. Then, if you need to use JWT tokens for your authentication, you must add the `shared-secret`.

You must enable and add the server certificate path to configure communication via HTTPS. You can see the required parameters in the figure below:

```

1  [http]
2  ...
3  auth-enabled = true # enable auth
4  log-enabled = true
5  write-tracing = false
6  pprof-enabled = true
7  pprof-auth-enabled = true
8  debug-pprof-enabled = false
9  ping-auth-enabled = true
10 https-enabled = true # HTTPS enable
11 https-certificate = "./influxdb.pem" # server-side certificate
12 shared-secret = "my super secret pass phrase" # JWT option

```

■ **Listing 3.18** The InfluxDB side security configuration

Now that our data is collected on monitoring servers and securely transferred to the collection server let us talk about the use of this data.

3.5 Visualization

In this chapter, we will get acquainted with the visualization of metrics and data that we have created using Grafana. We will talk about the structure of visualizations in Grafana, how panels are created, and also summarize what was visualized in this work.

3.5.1 Dashboards

The largest visualization unit in Grafana is the dashboard. The dashboard consists of panels, which can be charts, bar charts, indicators, and others. As an example, we provide the dashboard, which was designed to monitor the Docker containers for each specific hospital, so it contains metrics such as the number of running containers, container usage of CPU, memory, and many others. You can get acquainted with this dashboard in Appendix A.

Also, for the dashboard, variables can be defined. Those variables become global for the panels created in the dashboard. In our particular case, this is a `hospital` variable that filters information for each hospital.

We will not describe all dashboards in this way. Instead, we will provide a summary table to have an idea of the number of metrics that we are monitoring. The summary table is below:

Dashboard title	Panel title
Collection server monitoring	Etl jobs recieved number Application exceptions recieved number
Holt-Winters monitoring	Holt winters for cpu usage Holt winters for memory available
Hospital docker monitoring	Containers statuses Containers Block I/O Memory usage Container network tx CPU Usage
Host metrics monitoring	Disk usage in % Disk I/O read Disk I/O write CPU Usage Network bytes sends Network bytes recieved Memory available

■ **Table 3.1** Visualization summary table

Thanks to this table, you can see that, for the most part, the visualization covers all the basic metrics. Let us take a look at how the panels are created. To reduce the size of this chapter, we will only provide queries as examples. The full UI version of these queries will be added as Appendix B.

3.5.2 Panels

In this section, we will take a look at how panels are created. The most typical example, in this case, is the visualization of a specific metric that is stored in InfluxDB. Let's break down the visualization of CPU usage for each docker container:

```

1 SELECT mean("usage_percent")
2 FROM "docker_container_cpu"
3 WHERE ("hospital" =~ /^$hospital$/) AND $timeFilter
4 GROUP BY time($__interval), "container_name" fill(null)

```

■ **Listing 3.19** Grafana's panel creation InfluxQL example

InfluxQL is SQL-like. Its syntax is not so difficult to understand due to the similarity of constructions. Also, some constructions arise thanks to the Grafana, so that we will see them in classical SQL queries.

Typical for Grafana is the use of variables. Anything that starts with a \$ sign is a variable. For example, `$timeFilter` is the variable that is replaced with the currently selected panel time range. A rather complicated construction is used for global variables. For example, using the hospital variable shown earlier in the expression looks like this: `/^$hospital$/.`

This query returns for `docker_container_cpu` metric the percent CPU usage. It is filtered by the hospital and visualized for each Docker container.

The InfluxQL query looks very succinctly, since we are talking about the measurement. The peculiarity of the measurement is that all the necessary measurement data is already tied to the measurement time. For example, for each measurement, the hospital tag by which we filter is repeated. It does not save much memory, but the search for the information necessary for the question does not occur.

Now, let us look at an example of visualizing data received from a relational database. In this example, we will analyze the visualization of data about the ETL jobs that we receive from the monitoring server. To realize it, we create the query that looks next:

```

1 SELECT
2   $__timeGroupAlias(created_at,5m),
3   count(etl_job.id) AS "Etl jobs recieved for ${hospital}"
4 FROM etl_job
5 LEFT JOIN hospital ON etl_job.hospital_id = hospital.id
6 WHERE hospital.name = '${hospital}' AND $__timeFilter(created_at)
7 GROUP BY 1
8 ORDER BY 1

```

■ **Listing 3.20** Grafana’s panel creation SQL example

An important clarification is that for data visualization, an important condition is the binding of records to time. In the case of a time series database, each measurement is always tied to a time. However, in the case of a relational database, we need to understand what time attribute to use for visualization. In our case, the time of record creation is used.

This query returns the number of received job records for a particular hospital. Then this result is divided using a `$__timeGroupAlias` into intervals of 5 minutes and lined up in a histogram. Thus, the number is calculated for each of the histogram intervals.

In order not to confuse the reader, it should be noted that Grafana’s syntax varies depending on the query language. For example, the global variable in the case of SQL has the following syntax `'${hospital}'`. However, the context in which we use this variable does not change, so we hope that this fact will not cause difficulties.

For one panel, you can also create several questions. Let’s look at them in the example of predicting values using the Holt-Winters method. In this case, two charts are placed on one visualization panel, a metric chart, and a value prediction chart. I bring to your attention a chart of RAM prediction based on the Holt-Winters method:


```

1 # Holt-Winters prediction
2 SELECT holt_winters_with_fit(mean("available"),10,4)
3 FROM "mem"
4 WHERE $timeFilter AND "hospital" =~ /^$hospital$/
5 GROUP BY time($__interval), "hospital" fill(null)
6
7 # Memory available metric
8 SELECT mean("available")
9 FROM "mem"
10 WHERE $timeFilter AND "hospital" =~ /^$hospital$/
11 GROUP BY time($__interval), "hospital" fill(null)

```

■ **Listing 3.21** Grafana's panel creation multiquery example

This example is similar to the one we saw at the beginning of the section, but here, in addition to the metric itself, for its prediction Holt-Winters method is used. We also want to focus your attention to the fact that InfluxDB natively supports prediction using the Holt-Winters method that we will be used to create thresholds in the next section.

3.6 Failure prediction

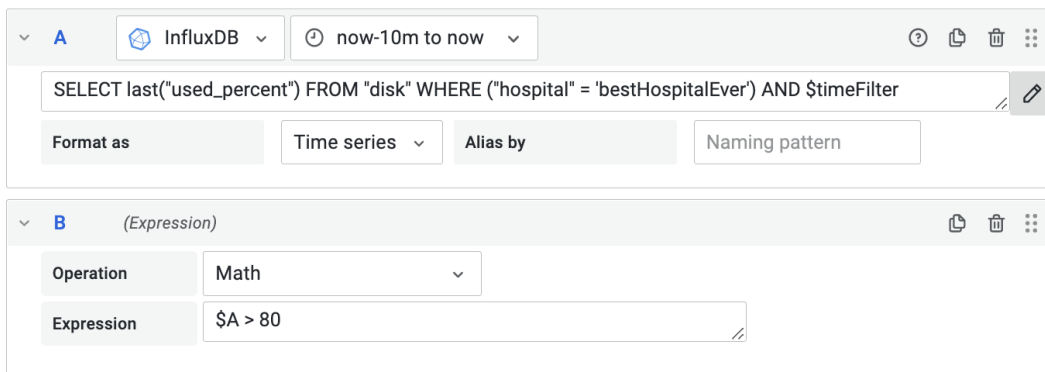
In this chapter, we will get acquainted with the failure prediction of metrics and data that we have realized using Grafana. In this section, we will analyze the failure prediction we have implemented and consider the prediction methods, that we used in this work. We used the static thresholds method and the Holt-Winters method for failure prediction.

Failure prediction in the Grafana is based on alerts firing. One of the important settings for the alert is when it will fire. The firing depends on the condition. If the condition is true, then Grafana will notify the user.

3.6.1 Thresholds method

In this section, we will talk about the implementation of the threshold method for Grafana. We will also look at an example of how the threshold is configured for Grafana's alerts.

Let us provide an example of alert settings based on disk space data. Below you can see two expressions, depending on which alert is firing:



■ **Figure 3.1** Thresholds method example

As you can see, the first expression is the InfluxDB query, similar to what we saw in the section before. This query gets the last disk space value. In our case, the query returns a single value, and we can use it in simple conditions. That is the reason why the second expression is the classic math condition. In this expression, we just get the result of query A and compare it with the constant. For now, if disk space is used more than 80% we will get the notification.

3.6.2 Holt-Winters seasonal method

So there are some cases where you can not compare your metric with the constant. For example, to detect some anomaly, you need to compare the measurement with some other value. That is the reason why we will use the Holt-Winters seasonal method. Let us provide an example of alert settings that detects network anomaly. Below you can see two expressions, depending on which alert is firing:

The screenshot shows a configuration interface for an alert with five expressions (A-E):

- Expression A:** InfluxDB query: `SELECT holt_winters_with_fit(difference(mean("bytes_recv")),10,4) FROM "net" WHERE $timeFilter GROUP BY time($__interval), "hospital" fill(null)`. Format: Time series.
- Expression B:** InfluxDB query: `SELECT difference(mean("bytes_recv")) FROM "net" WHERE $timeFilter GROUP BY time($__interval), "hospital" fill(null)`. Format: Time series.
- Expression C:** Operation: Reduce. Function: Last. Input: A. Mode: Strict.
- Expression D:** Operation: Reduce. Function: Last. Input: B. Mode: Strict.
- Expression E:** Operation: Math. Expression: `abs($B*3)+1000 < abs($A)`.

■ **Figure 3.2** Holt-Winters seasonal method example

Let us explain how these expressions work together. First of all, the monitored metric is the number of received bytes. This metric is cumulative. To work with the single value, we will take the measurement cumulation. That is the reason why we call the `difference()` function. Also, we create a similar expression for Holt-Winters prediction.

For now, A and B expressions get sets of measurements. To get the single value, we need to reduce both sets and apply the reduce function. As you can see, C and D expressions return only the last values of sets that we have before.

Finally, we can create a simple condition that returns true or false. In our case, the condition compares the real value with the expected value. And if this value is very different from the expected one, we will receive a notification that the anomaly has been detected.

Of course, more complex methods, such as Holt-Winters, need to be tested on real data. The constants need to be obtained after some experiments.

We have no access to the real data in this work, and the main goal of this method is to demonstrate how it should work for real cases, with little modification of constants.

3.7 Implementation summary

In this section, the implementation will be summarized. First of all, we create the work solution that can be used in production after some administration details. For example, generate real production certificates.

This solution realizes all requirements that we define in the requirements analysis. In the future, some sections, such as visualization and error prediction, may be added as needed. The work has been implemented in the volume required by the Alpha organization. In the next chapter, we provide information about testing the logic we have implemented.

Chapter 4

Testing

In this chapter, we will describe how the final solution was tested. We will describe in detail what kind of test we have done and what components were tested. Also, we will provide information about the test coverage.

4.1 Integration tests

In this section, we will focus on integration tests. Let us describe in detail what components we tested and what techniques we used when writing integration tests.

As you know, integration tests are defined as testing in which program modules are logically integrated and tested as a group. This testing level aims to identify defects in the interaction between software modules.

In our case, integration tests were created for two components that we implemented of our solution: collection and monitoring servers. Let us look at the features of their testing.

4.1.1 Monitoring server integration tests

This section will examine the integration tests used to test the monitoring server. On the monitoring server side, much essential logic deserves detailed testing. For this, integration tests of two types were carried out:

- Synchronization tests
- Batch tests

Let us look at the characteristics of each type.

Synchronization tests

The synchronization tests' purpose was to determine if the collection server's sync process was working correctly. For this, the Mocking technique was applied, and a mock object was created that simulate the behavior of the collection server, saving data and behavior in various modes of operation: normal, turned off, and returning errors only. You can see the code example of the collection server mock below:

```

1  class DummyCommunicationBusiness : ICommunicationBusiness {
2
3      var db = mutableListOf<Synchronizable>()
4      var mode = DummyCommunicationBusinessMode.NORMAL
5
6      override fun pushBatch(batch: List<Synchronizable>): State {
7          applyMods()
8          db.addAll(batch)
9          if (db.isEmpty()){
10             return State(stateName = StateName.NOT_INIT)
11         }
12         return State(db.sortedWith(compareBy{it.createdAt}).last().monitoringId)
13     }
14     override fun pullState(): State {
15         applyMods()
16         if (db.isEmpty()){
17             return State(stateName = StateName.START)
18         }
19         return State(db.sortedWith(compareBy{it.createdAt}).last().monitoringId)
20     }
21     fun applyMods(){
22         if (mode == DummyCommunicationBusinessMode.BAD){
23             throw SessionInterruptedException()
24         }
25         if (mode == DummyCommunicationBusinessMode.TURNED_OFF){
26             throw SessionInterruptedException()
27         }
28     }
29 }

```

■ **Listing 4.1** The mock of collection server

As you can see from the example code, the mock object provides the most similar behavior compared to the real collection server. This mock object allows us to test quite complex cases in the synchronization process between monitoring and collection services.

Let us provide an example of synchronization testing using this mock service in the following figure:

```

1  @Test
2  @DirtiesContext(classMode = DirtiesContext.ClassMode.BEFORE_EACH_TEST_METHOD)
3  fun syncOneTest() {
4      assert(dummyCommunicationBusiness.db.isEmpty())
5      assert(appSyncBusiness.isStateNotInit())
6      appExceptionRepository.save(AppException())
7      syncTasks.syncTask()
8      assert(!appSyncBusiness.isStateNotInit())
9      assert(dummyCommunicationBusiness.db.count() == 1)
10 }

```

■ **Listing 4.2** Basic integration test for synchronization process

Let us describe the code example. At the beginning of this test, the database of `dummyCommunicationBusiness`, which simulates the collection server, is empty. Then, we save the `AppException` instance in the monitoring database and execute the synchronization task. As you can see, the `AppException` record was added to the collection database.

The `dummyCommunicationBusiness` also has mods that allow us to simulate the collection server problems. Such as failures and communication errors. In the following code example, you can see the synchronization testing with modes applying:

```

1  ...
2  assert(!appSyncBusiness.isStateNotInit())
3  assert(dummyCommunicationBusiness.db.count() == 1)
4
5  // now bad response only
6  dummyCommunicationBusiness.mode = DummyCommunicationBusinessMode.BAD
7  appExceptionRepository.save(AppException())
8  syncTasks.syncTask()
9  // communication refresh
10 assert(appSyncBusiness.isStateNotInit())
11 // no sync happened
12 assert(dummyCommunicationBusiness.db.count() == 1)
13 syncTasks.syncTask()
14 // communication refresh again
15 assert(appSyncBusiness.isStateNotInit())
16 assert(dummyCommunicationBusiness.db.count() == 1)
17
18 // normal communication again
19 dummyCommunicationBusiness.mode = DummyCommunicationBusinessMode.NORMAL
20 // it works again
21 syncTasks.syncTask()
22 assert(!appSyncBusiness.isStateNotInit())
23 assert(dummyCommunicationBusiness.db.count() == 2)
24 ...

```

■ **Listing 4.3** The integration test for synchronization process failure

At the beginning of this test fragment, the synchronization worked normally. Then, after mode switching, synchronization was unsuccessful, and the synchronization refresh happened, and the synchronization state became `NOT_INIT`. After that, we switched the mode back, and the record was synchronized correctly.

We have achieved sufficiently detailed testing of the synchronization-related logic and can guarantee that the synchronization occurs as we expected.

Batch tests

The purpose of the batch tests was to determine if the batches were generated correctly for each kind of data. These tests are not particularly complex but also essential and required.

For example, it tests the batch items order, which you can see in the next code example:

```

1  @Test
2  @ DirtiesContext(classMode = DirtiesContext.ClassMode.BEFORE_EACH_TEST_METHOD)
3  fun orderInBatchStartTest() {
4      appSyncBusiness.updateState(State(stateName = StateName.START))
5      val a1 = AppException(createdAt = Instant.now())
6      val a2 = AppException(createdAt = Instant.now().plusSeconds(10))
7      // save in inverse order
8      appExceptionRepository.save(a2)
9      appExceptionRepository.save(a1)
10     // is last element correct
11     assert(appSyncBusiness.createBatch().last().monitoringId == a2.monitoringId)
12 }

```

■ **Listing 4.4** Basic integration test for batch creation process

As you can see, this test example creates items with different creation date and save them in inverse order, but in batch, the last one is the record with the newest creation date.

For now, let us provide information about the tests carried out on the side of the collection server.

4.1.2 Collection server integration tests

This section will look at the integration tests used to test the collection server.

It is important to note that the collection server does not contain complex logic that deserves testing. However, it is necessary for us to test the interaction of all application modules.

Therefore, to test the collection server, we use `MockMvc` — a special Spring class that allows you to simulate the receiving of a specific request by the controller. It gives us the ability to test individual application endpoints.

In the following code example, you can see the collection server testing with `MockMvc` applying:

```

1  fun pushBatch(){
2      hospitalRepository.save(Hospital(name = hospitalName))
3      val batchFirst = AppException()
4      val batchLast = AppException()
5      val batch = listOf(batchFirst, batchLast)
6      val result = mockMvc.perform(post("$url/batch")
7          .accept(MediaType.APPLICATION_JSON)
8          .contentType(MediaType.APPLICATION_JSON)
9          .content(mapper.writeValueAsString(SyncRequest(batch, hospitalName))))
10     .andExpect(status().isOk)
11     .andExpect(content().contentType(MediaType.APPLICATION_JSON))
12     .andReturn()
13     assert(result.response.contentAsString == mapper.writeValueAsString(
14         State(batchLast.monitoringId, StateName.IN_PROCESS)))
15     assert(batchLast.monitoringId ==
16         appRepository.getLastMonitoringId(hospitalName))
17     assert(appRepository.findAll().count() == batch.count())
18 }

```

■ **Listing 4.5** The collection solution testing using the `MockMvc`

In this code example, the `mockMvc` simulates the batch receiving from the monitoring server. Then, we compare the batch we created at the test and sent to the collection server endpoint, using `mockMvc` with the batch we stored.

As you can see, this kind of test covers all major parts of the application architecture, such as persistence, domain logic, and the presentation layer. Thus, the integration tests with `mockMvc` tests the interaction of all components in the application with a relatively small number of tests.

4.2 Tests coverage


One of the important indicators of testing is test coverage. It allows us to identify the code that did not cover by tests yet. In this section, we will turn your attention to the test coverage of two applications that we have implemented in this work, namely the collection and the monitoring servers.

There is the collection server test coverage in the figure below:

Overall Coverage Summary

Package	Class, %	Method, %	Line, %
all classes	92.9% (26/28)	89.5% (51/57)	88.6% (140/158)

Coverage Breakdown

Package 	Class, %	Method, %	Line, %
com.collectionserver	50% (1/2)	50% (1/2)	50% (1/2)
com.collectionserver.configuration	66.7% (2/3)	50% (4/8)	41.7% (10/24)
com.collectionserver.controller	100% (3/3)	100% (8/8)	100% (8/8)
com.collectionserver.model.entity	100% (4/4)	90.9% (10/11)	98.2% (54/55)
com.collectionserver.model.enums	100% (1/1)	100% (1/1)	100% (1/1)
com.collectionserver.model.vo.communication	100% (5/5)	100% (6/6)	100% (10/10)
com.collectionserver.model.vo.synchronization	100% (1/1)	100% (1/1)	100% (3/3)
com.collectionserver.repository.custom	100% (2/2)	100% (4/4)	87.5% (14/16)
com.collectionserver.service	100% (3/3)	100% (9/9)	100% (30/30)
com.collectionserver.validation	100% (1/1)	100% (4/4)	100% (6/6)
com.monitoringserver.model.enums	100% (3/3)	100% (3/3)	100% (3/3)

■ **Figure 4.1** The collection server tests coverage

As you can see, most of the code is covered by tests. The only exception is the configuration, which has some uncovered cases in which it is unnecessary to test. Thus, we can confidently say that we have tested the collection server code in detail.

Let us also pay attention to the monitoring server test coverage. In the following picture, you can see the monitoring server:

Overall Coverage Summary

Package	Class, %	Method, %	Line, %
all classes	89.7% (35/39)	79.5% (70/88)	80% (180/225)

Coverage Breakdown

Package ▾	Class, %	Method, %	Line, %
com.monitoringserver.validation	100% (1/1)	100% (4/4)	100% (6/6)
com.monitoringserver.service	100% (2/2)	50% (2/4)	71.4% (5/7)
com.monitoringserver.model.vo.synchronization	100% (2/2)	100% (5/5)	100% (8/8)
com.monitoringserver.model.vo.communication	100% (3/3)	100% (3/3)	100% (5/5)
com.monitoringserver.model.factory	100% (1/1)	100% (4/4)	100% (4/4)
com.monitoringserver.model.enums	100% (4/4)	100% (4/4)	100% (4/4)
com.monitoringserver.model.entity	100% (3/3)	100% (6/6)	100% (39/39)
com.monitoringserver.exception.synchronization	100% (2/2)	100% (2/2)	100% (2/2)
com.monitoringserver.exception	100% (1/1)	100% (1/1)	100% (1/1)
com.monitoringserver.controller	100% (4/4)	36.4% (4/11)	47.1% (8/17)
com.monitoringserver.configuration	50% (2/4)	55.6% (5/9)	40% (10/25)
com.monitoringserver.components.task	75% (3/4)	50% (4/8)	55.6% (20/36)
com.monitoringserver.business.synchronization	100% (3/3)	100% (14/14)	96.8% (30/31)
com.monitoringserver.business.communication	100% (3/3)	100% (11/11)	97.4% (37/38)
com.monitoringserver	50% (1/2)	50% (1/2)	50% (1/2)

■ **Figure 4.2** The monitoring server tests coverage

In the case of the monitoring server, test coverage is somewhat less. It is because the monitoring server at the time of this writing has not yet been integrated with the hospital application.

Because of this, the controller and services have been tested somewhat less than the rest of the application. While we currently cannot complete the integration, we expect tests will be written when the integration is complete.

Just like in the case of the collection server, some configuration that did not require testing was not covered by the tests.

Otherwise, we can confidently say that all the logic used in our solution has been thoroughly tested and works as expected.

4.3 Testing summary

In this section, we got acquainted with the use of integration tests in practice. We also got acquainted with some auxiliary techniques used in testing, such as mock creating.

Integration tests are very important for testing an application, and thanks to them, you can test the application logic to the fullest.

We could also measure, using test coverage, how detailed our code was tested. Although test coverage is a very important indicator, it does not mean that the application has been fully tested so that these tests may be expanded in the future.

We can declare that the main logic of the application has been tested in detail, and all tested processes work as intended. Now that our work is ready let's summarize it and suggest possible future improvements.

Conclusion

The aim was to design and implement a solution that allows observing hospital servers' work from the outside. To do this, we studied an existing solution and got acquainted with the requirements for a future solution. We also explore the environment in which the future solution was supposed to work.

Having received an idea about what the future solution must be, we studied the information necessary to create it. We learned domains related to metrics collection, security, error prediction, and others to design a solution. In the end, we designed and implemented the final solution and then carried out the necessary testing.

We can claim that we have achieved the originally intended goal based on the work carried out. We created a solution that meets all the requirements and can be used for its intended purposes.

We can confidently say that the work is finished. Now, we will state some improvements that should be added in the future, if the Alpha company considers it necessary for their product.

The first possible improvement that will bring a lot to this solution is the replication of the collection solution. In our version, the collection solution can become a bottleneck, and if it falls, monitoring becomes impossible.

The next improvement is the monitoring metrics on the side of the collection solution. It would also be a very important addition to the infrastructure. Thanks to this, we will also be able to predict failures on the side of the collection solution and reduce the amount of time in which the collection solution will be unavailable.

It should also be mentioned that it is worth adjusting the alert rules over time. At this stage, we still need long-term statistics in order to decide how effective the rules we have created are. However, over time it will become clear what adjustments should be realized to make these rules more relevant.

We want to note that we do not use all the collected metrics. Therefore, in the future, it is worth considering supplementing alert rules and dashboards with metrics that have not yet been used.

It is not a whole list of improvements that can be implemented in this work. I hope, that some of them, I will be able to implement after my master's study ends.

..... Appendix A

Grafana dashboard examples

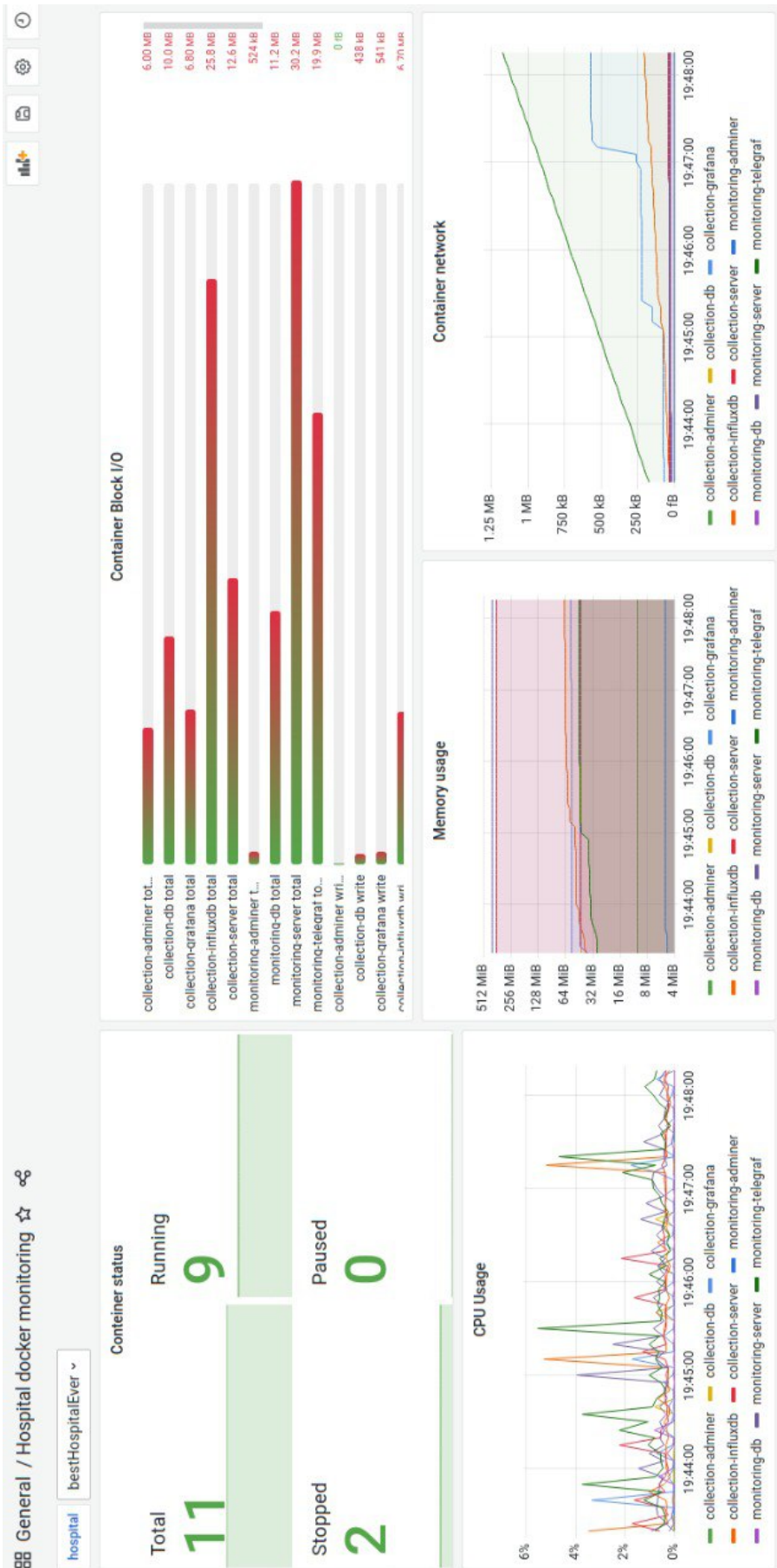
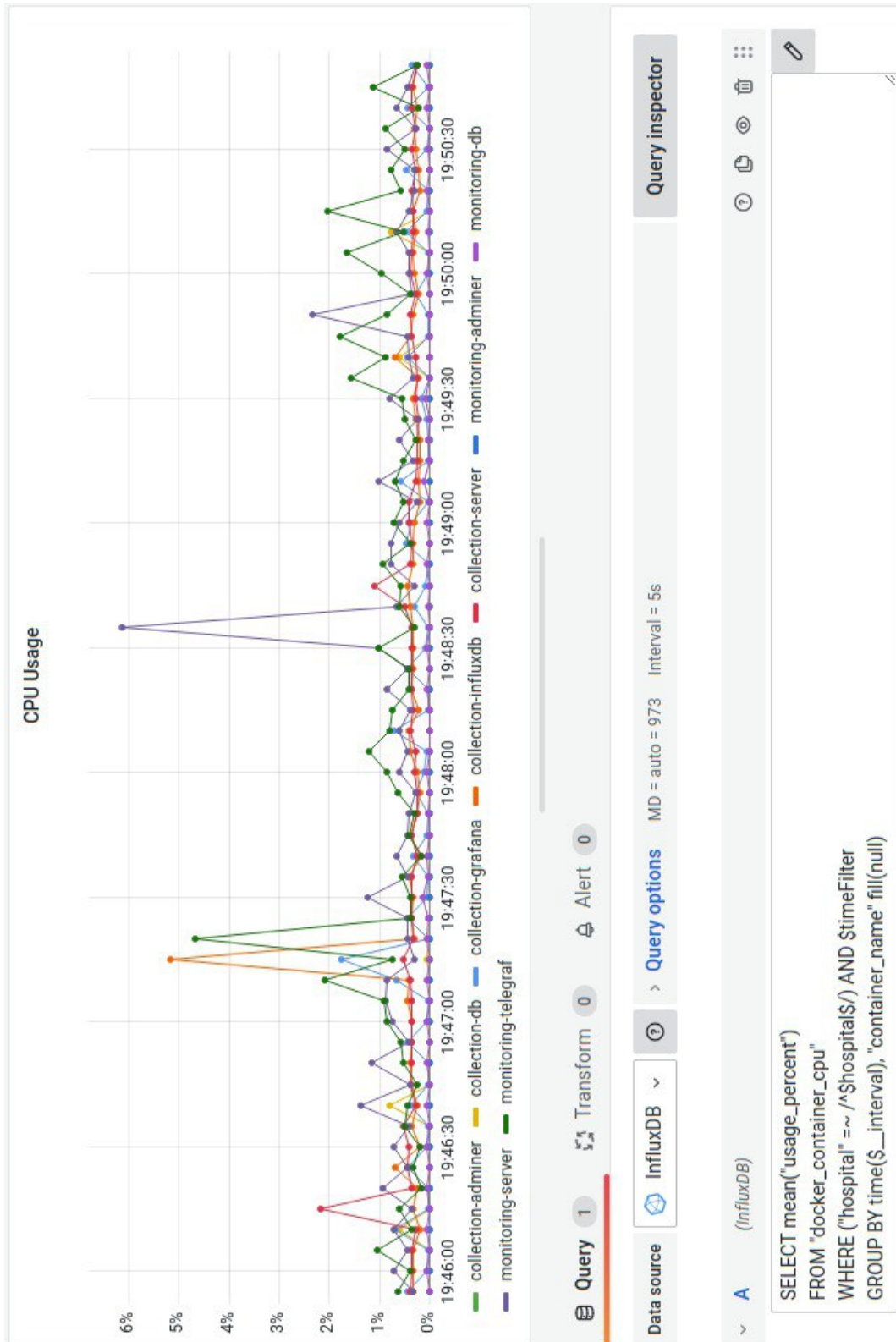


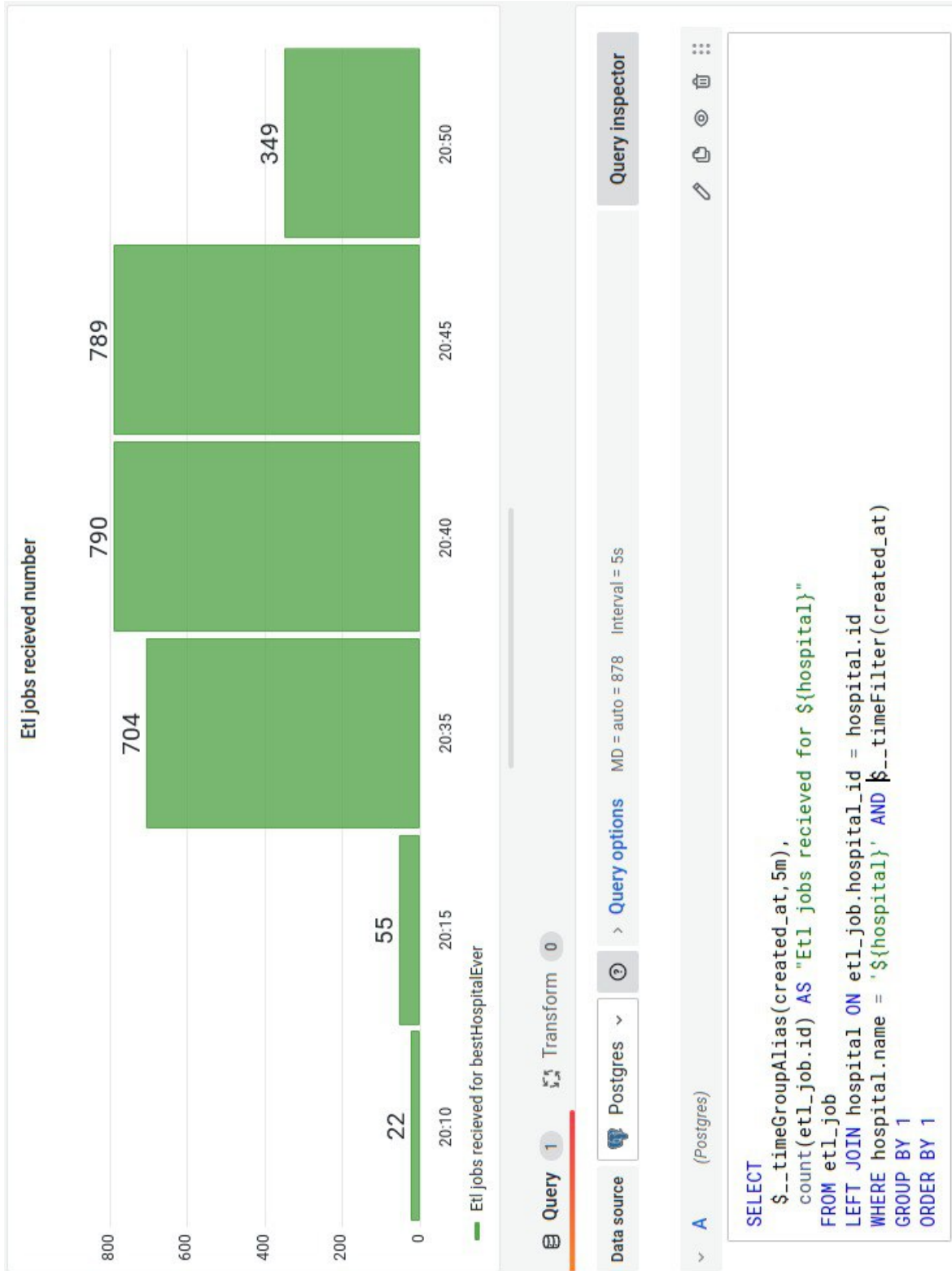
Figure A.1 The hospital Docker monitoring dashboard

..... Appendix B

Grafana panel examples



■ Figure B.1 CPU usage panel creation example



■ Figure B.2 ETL jobs received panel creation example



■ Figure B.3 Holt-Winters based panel creation example

Bibliography

1. INC., Docker. *docker stats* — *Docker Documentation [online]*. Available also from: <https://docs.docker.com/engine/reference/commandline/stats/>.
2. INC., Docker. *Runtime metrics* — *Docker Documentation [online]*. Available also from: <https://docs.docker.com/config/containers/runmetrics/>.
3. INC., Docker. *Engine API v1.21* — *Docker Documentation [online]*. Available also from: <https://docs.docker.com/engine/api/v1.21/>.
4. PROMETHEUS AUTHORS. *Querying basics* — *Prometheus [online]*. Available also from: <https://prometheus.io/docs/prometheus/latest/querying/basics/>.
5. INC., Docker. *Docker overview* — *Docker Documentation [online]*. Available also from: <https://docs.docker.com/get-started/overview/>.
6. MENAGE, Paul. *CGROUPS [online]*. Available also from: <https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt>.
7. YOUNG, K. *How to Collect Docker Metrics* — *Datadog [online]*. Available also from: <https://www.datadoghq.com/blog/how-to-collect-docker-metrics/>.
8. YOUNG, K. *How to Collect Docker Metrics* — *Datadog [online]*. Available also from: <https://www.datadoghq.com/blog/how-to-collect-docker-metrics/>.
9. DEVELOPMENT COMMUNITY, The kernel. *The /proc Filesystem* — *The Linux Kernel documentation [online]*. Available also from: <https://www.kernel.org/doc/html/latest/filesystems/proc.html>.
10. LLC, Google. *google/cadvisor: Analyzes resource usage and performance characteristics of running containers [online]*. Available also from: <https://github.com/google/cadvisor>.
11. AUTHORS, Prometheus. *Overview* — *Prometheus [online]*. Available also from: <https://prometheus.io/docs/introduction/overview/>.
12. PROMETHEUS AUTHORS. *Exporters and integrations* — *Prometheus [online]*. Available also from: <https://prometheus.io/docs/instrumenting/exporters/>.
13. INC., Docker. *dockerd* — *Docker Documentation [online]*. Available also from: <https://docs.docker.com/engine/reference/commandline/dockerd/#daemon-metrics>.
14. PROMETHEUS AUTHORS. *Storage* — *Prometheus [online]*. Available also from: <https://prometheus.io/docs/prometheus/latest/storage/>.
15. PROMETHEUS AUTHORS. *Querying basics* — *Prometheus [online]*. Available also from: <https://prometheus.io/docs/prometheus/latest/querying/basics/>.
16. INC., InfluxData. *InfluxDB 1.X: Open Source Time Series Platform* — *InfluxData [online]*. Available also from: <https://www.influxdata.com/time-series-platform/>.

17. CLOUDBEES, Inc. *Infrastructure Monitoring with TICK Stack — Cloudbees Blog [online]*. Available also from: <https://www.cloudbees.com/blog/infrastructure-monitoring-with-tick-stack>.
18. INC., InfluxData. *InfluxDB 1.X: Open Source Time Series Platform — InfluxData [online]*. Available also from: <https://www.influxdata.com/time-series-platform/>.
19. INC., InfluxData. *The plugin-driven server agent for collecting and reporting metrics [online]*. Available also from: <https://github.com/influxdata/telegraf/tree/master/plugins/outputs>.
20. INC., InfluxData. *Docker Input Plugin [online]*. Available also from: <https://github.com/influxdata/telegraf/blob/master/plugins/inputs/docker/README.md>.
21. INC., InfluxData. *InfluxDB 1.X: Open Source Time Series Platform — InfluxData [online]*. Available also from: <https://www.influxdata.com/time-series-platform/>.
22. INC., InfluxData. *Chronograf: Complete Dashboard Solution for InfluxDB [online]*. Available also from: <https://www.influxdata.com/time-series-platform/chronograf/>.
23. INC., InfluxData. *Kapacitor and Real-Time Stream Processing — InfluxDat [online]*. Available also from: <https://www.influxdata.com/time-series-platform/kapacitor/>.
24. BORKO FURHT Edin Muharemagic, Daniel Socek. *Multimedia Encryption and Watermarking [online]*. Available also from: https://link.springer.com/chapter/10.1007/978-3-319-26090-0_3.
25. OKTA. *Authentication vs. Authorization — Okta [online]*. Available also from: <https://www.okta.com/identity-101/authentication-vs-authorization/>.
26. LIMITED, Neumetrix. *HTTP Authentication — HttpWatch [online]*. Available also from: <https://www.httpwatch.com/httpgallery/authentication/>.
27. SEVCSIK-ZAJÁ CZ, Andras. *Authentication using HTTPS client certificates [online]*. Available also from: <https://medium.com/@sevcsik/authentication-using-https-client-certificates-3c9d270e8326>.
28. D. HARDT, Ed. *RFC 6749: The OAuth 2.0 Authorization Framework [online]*. Available also from: <https://www.rfc-editor.org/rfc/rfc6749>.
29. SEVCSIK-ZAJÁ CZ, Andras. *Authentication using HTTPS client certificates [online]*. Available also from: <https://medium.com/@sevcsik/authentication-using-https-client-certificates-3c9d270e8326>.
30. SEVCSIK-ZAJÁ CZ, Andras. *Authentication using HTTPS client certificates [online]*. Available also from: <https://cogitogroup.net/blog/2019/11/29/pki-the-pros-and-cons/>.
31. INC, Stack Exchange. *Information Security [online]*. Available also from: <https://security.stackexchange.com/questions/198837/why-is-client-certificate-authentication-not-more-common?rq=1>.
32. GRANT KELLY, Bruce McKenzie. *Security, privacy, and confidentiality issues on the Internet [online]*. Available also from: <https://www.ncbi.nlm.nih.gov/pmc/articles/PMC1761937/>.
33. FARAHANI, Shahin. *Message Integrity Code [online]*. Available also from: <https://www.sciencedirect.com/topics/computer-science/message-integrity-code>.
34. LABS, Grafana. *Message Integrity Code - an overview [online]*. Available also from: <https://grafana.com/docs/grafana/latest/datasources/>.
35. PROMETHEUS AUTHORS. *Grafana — Prometheus [online]*. Available also from: <https://prometheus.io/docs/visualization/grafana/>.

36. RAMACHANDRAN, Mahesh. *Anatomy Of An IT Outage: Prediction and Detection [online]*. Available also from: <https://blog.opsramp.com/it-outage-prediction-and-detection>.
37. DATE, Sachin. *Time Series Analysis, Regression and Forecasting[online]*. Available also from: <https://timeseriesreasoning.com/contents/holt-winters-exponential-smoothing/>.
38. INC., InfluxData. *Functions — InfluxData Documentation Archive [online]*. Available also from: https://archive.docs.influxdata.com/influxdb/v1.0/query_language/functions/#holt-winters.
39. HYNDMAN, Rob J; ATHANASOPOULOS, George. *Non-seasonal ARIMA models — Forecasting: Principles and Practice (2nd ed)[online]*. Available also from: <https://otexts.com/fpp2/non-seasonal-arima.html>.
40. NARIZHNYKH, Dmitry. *What is Database Synchronization?[online]*. Available also from: <https://dbconvert.com/blog/what-is-database-synchronization/>.
41. *PYPL PopularitY of Programming Language [online]*. Available also from: <https://pypl.github.io/PYPL.html>.
42. BANSAL, Anshul. *Java vs. Kotlin — Baeldung on Kotlin [online]*. Available also from: <https://www.baeldung.com/kotlin/java-vs-kotlin#:~:text=Java%20is%20a%20strictly%20typed,type%20of%20the%20assignment%20value>.
43. HARTMAN, James. *Kotlin vs Java - Difference Between Them [online]*. Available also from: <https://www.guru99.com/kotlin-vs-java-difference.html#9>.
44. INC., Docker. *Collect Docker metrics with Prometheus [online]*. Available also from: <https://docs.docker.com/config/daemon/prometheus/>.
45. PROMETHEUS AUTHORS. *Monitoring Docker container metrics using cAdvisor — Prometheus [online]*. Available also from: <https://prometheus.io/docs/guides/cadvisor/>.
46. LABS, Grafana. *Play with Grafana Mimir [online]*. Available also from: <https://grafana.com/tutorials/play-with-grafana-mimir/?pg=oss-mimir&plcmt=hero-btn-1>.
47. INC., Docker. *Use Docker Compose — Docker Documentation [online]*. Available also from: https://docs.docker.com/get-started/08_using_compose/#:~:text=Docker%20Compose%20is%20a%20tool,or%20tear%20it%20all%20down..
48. LABS, Grafana. *Alerting provisioning API [online]*. Available also from: https://grafana.com/docs/grafana/v9.0/developers/http_api/alerting_provisioning/.

Contents of enclosed media

	readme.txt	the file with CD contents description	
	src	the directory of source codes	
		impl	implementation source codes
		thesis	the directory of L ^A T _E X source codes of the thesis
	text	the thesis text directory	
		thesis.pdf	the thesis text in PDF format