TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Aleksandr Borovkov 206339IAAB

Development of a Standardized Set of Checks for Apps in OpenShift

Bachelor's thesis

Supervisor: Aleksei Talisainen,
MSc

Tallinn 2023

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Aleksandr Borovkov 206339IAAB

OpenShiftis rakenduste jaoks standardiseeritud kontrollide komplekti väljatöötamine

Bakalaureusetöö

Juhendaja: Aleksei Talisainen,
MSc

Tallinn 2023

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Aleksandr Borovkov

24.04.2023

Abstract

Development of a Standardized Set of Checks for Apps in OpenShift

The goal of this study is to minimize the risk of downtime and other application failures in the OpenShift environment by proactively identifying and resolving potential application problems by developing a monitoring solution. In addition to this, a number of tests will be carried out, confirming the effectiveness of this solution. In the theoretical part, the author analyzes the main problems associated with the lack of system monitoring for applications in the OpenShift environment, and also explains the most important aspects of monitoring and various solutions in this area. The practical part of the work is devoted to the development of a monitoring solution for a real OpenShift project, which includes several container applications.

The chosen topic of the thesis is relevant for several reasons. Firstly, OpenShift is a containerization and orchestration platform that allows to develop, deploy, and scale applications in containers. Due to the growing popularity of container technologies and the development of cloud computing, OpenShift is increasingly popular among developers.

Secondly, applications deployed on the OpenShift platform can contain many components and dependencies, making them vulnerable to various security threats. Developing a standardized control set for applications in OpenShift can help prevent various vulnerabilities and provide a higher level of security.

And the last reason is that a standardized set of controls can simplify the process of developing and testing applications on the OpenShift platform. Developers can use the set of checks as a guide to ensure their applications meet quality requirements.

This thesis is written in English and is 31 pages long, including 5 chapters, 11 figures and 2 tables.

Key words: monitoring, cybernetic, notification systems.

Annotatsioon

OpenShiftis rakenduste jaoks standardiseeritud kontrollide komplekti väljatöötamine

Selle uuringu eesmärgiks on minimiseerida seisakute ja muude rakenduste tõrgete riski OpenShift keskkonnas, tuvastades ja lahendades ennetavalt võimalikud rakenduse probleemid ning töötades välja jälgimislahenduse. Lisaks sellele viiakse läbi mitmeid teste, mis kinnitavad selle lahenduse tõhusust. Teoreetilises osas analüüsib autor peamisi probleeme, mis on seotud OpenShift keskkonnas olevate rakenduste süsteemiseire puudumisega ning selgitab ka selle valdkonna jälgimise olulisemaid aspekte ja erinevaid lahendusi. Töö praktiline osa on pühendatud reaalse OpenShift projekti seirelahenduse väljatöötamisele, mis hõlmab mitmeid konteinerrakendusi

Lõputöö valitud teema on asjakohane mitmel põhjusel. Esiteks, OpenShift on konteineriseerimis- ja orkestreerimisplatvorm, mis võimaldab konteinerites arendada, juurutada ja skaleerida rakendusi. Konteinertehnoloogiate populaarsuse kasvu ja pilvandmetöötluse arengu tõttu on OpenShift arendajate seas üha populaarsem.

Teiseks, OpenShifti platvormil juurutatud rakendused võivad sisaldada palju komponente ja sõltuvusi, muutes need haavatavaks erinevate turvaohtude suhtes. Rakenduste jaoks standardiseeritud juhtkomplekti väljatöötamine OpenShiftis võib aidata vältida erinevaid haavatavusi ja pakkuda kõrgemat turbetaset.

Viimane põhjus on see, et standardiseeritud juhtelementide komplekt võib lihtsustada OpenShifti platvormil rakenduste arendamise ja testimise protsessi. Arendajad saavad kasutada kontrollide komplekti juhendina, et tagada nende rakenduste vastavus kvaliteedinõuetele.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 31 leheküljel, 5 peatükki, 11 joonist, 2 tabelit.

Märksõnad: monitooring, küberneetika, teavitussüsteemid.

List of abbreviations and terms

CPU	*Central processing unit*
Git	*Free and open source distributed version control system*
Init	*Initialization*
Src	*Source*
URL	*Uniform Resource Locator*
YAML	*Human-readable data serialization language*

Table of contents

List of figures

List of tables

1 Introduction

Developing a standardized set of controls for applications in OpenShift is an important topic that solves a common problem in the software development industry. Many developers face the challenge of ensuring the reliability and stability of their applications on the OpenShift platform, which can be a time-consuming and complex process.

As a result, this thesis was written with the aim of proposing a solution that can simplify and streamline this process. By developing a standardized control set for applications in OpenShift, developers can ensure that their applications are thoroughly tested and meet the required quality standards.

This approach offers several advantages, including better reliability, less downtime, and faster time-to-market. With a standardized set of checks, developers can identify potential problems early in development and fix them before they problems become bigger.

In addition, this work can benefit many stakeholders, including DevOps engineer, project managers, and companies than uses in their daily work OpenShift. DevOps engineers can save time and effort by using a standardized set of controls, and project managers can ensure a smooth and efficient development process. Businesses benefit from increased reliability and stability of their applications, which can increase customer satisfaction and profits.

In conclusion, developing a standardized set of controls for OpenShift applications is an important topic that can help solve a common problem in software development. By streamlining the testing process and ensuring application reliability and stability, DevOps engineers, project managers and entities benefit from increased efficiency, reduced downtime and increased customer satisfaction. In addition, a standardized control set can be useful in creating uniform quality standards in the software development industry, which in turn can improve product quality and increase the competitiveness of companies. All in all, developing a standardized control set for applications in OpenShift can have a significant positive impact on various aspects of software development and operation.

2 Analysis

2.1 Vision and the scope

One of the most important aspects of improving the application development and deployment process on the OpenShift platform is to develop a complete set of controls for the application, depending on the various aspects of the application. [1] OpenShift is a container platform that allows developers to deploy their own containerized applications. The OpenShift platform is based on the Docker container technology, [2] as well as the Kubernetes orchestrator. [3]

When developing applications on OpenShift, it is important that they meet standards and requirements such as performance, scalability, security and reliability. This requires a series of checks to ensure that applications run correctly and efficiently on the OpenShift platform. [1]

However, application verification in OpenShift can be problematic. Different organizations and teams may have different approaches to application development and deployment, and the lack of common standards can lead to inconsistencies, errors, and problems in the application deployment and operation process.

The author of this thesis has a clear vision of what kind of system monitoring and notification system should be implemented in the OpenShift environment to ensure stable and uninterrupted operation of applications. The main goal is to identify and fix problems in real time before they cause applications to fail.

One of the most important aspects to monitor is the state of applications and their restarts. If an application stops working, it can cause serious problems such as data loss, unavailability of services, and as a result, damage to the business. Therefore, monitoring the status of applications and restarting them is a must for reliable operation.

The author also considers it important to monitor the use of resources such as CPU and memory to prevent problems associated with their scarcity. In addition, monitoring the

availability of network connections and external services helps quickly identify problems in the application and take the necessary measures.

Thus, the main purpose of monitoring and alerting in OpenShift environment is to ensure that applications run reliably and minimize downtime in the event of problems. To do this, it is necessary to monitor various aspects such as application status, resource usage and take action to eliminate them in real time.

2.2 Methodology and used tools

2.2.1 Approach and standards

Effective application monitoring in OpenShift is essential for optimal performance and availability. Several approaches and standards can be used to achieve this goal. An important aspect of effective monitoring is starting from a solid foundation that includes a well-thought-out architecture, clear metrics, [4] and appropriate logging. [5] This helps to create a strong baseline and understand the normal operating parameters of the application.

Another important aspect of effective monitoring is using a monitoring tool that integrates well with OpenShift. [6] It allows to get real-time information about the performance and health of applications, which helps to identify problems and act quickly. Alarm and notification messages are also important for effective monitoring. Setting up alarms and notifications that meet the specific needs of application will help notify DevOps team immediately when something goes wrong. [7]

Monitoring as code enforcement is another important aspect of effective monitoring. Using tools such as Ansible, [8] Puppet, [9] Chef, [10] or Kustomize [11] to automate the deployment and configuration of monitoring tools ensures consistency and repeatability, making the monitoring solution easier to manage over time.

Finally, it's important to constantly monitor and evaluate application's performance and make adjustments as needed to ensure optimal performance and availability.

2.2.2 Critical check and alerts

Critical checks and alerts are important aspects of any resource monitoring and management system. They make it possible to notify responsible persons about important events that require their attention and response. [12]

Critical checks can be configured to monitor various parameters such as server availability, CPU usage, and memory usage. If problems occur during these checks, the system sends an alert to the responsible parties so that they can immediately react and solve the problem. [13] The author of the thesis uses a team of DevOps engineers as receivers, because in the case of an application failure, they can immediately start maintaining the system and application.

Also, an important element of the monitoring system is the prioritization of alerts, which makes it possible to determine how critical the problem is. If there are many alerts coming in at once, it makes sense to prioritize them in order to speed up the response to the most important ones. [14]

Alerts can be prioritized based on a number of factors, including task importance, urgency, potential consequences, impact on business processes, and more. Alerts are typically divided into four priority levels: [14]

- Critical is the highest priority level and indicates that there are events that require immediate attention and resolution. Such warnings should be sent immediately and attract the maximum attention of those responsible.

- High - Alerts at this priority level require quick response and action, but are not so critical as to require immediate attention. Such alerts should be sent within minutes of the event occurring.

- Medium - Alerts at this priority level indicate events that require attention but are not critical or high. They can be sent within an hour of the event.

- Low - Alerts at this priority level are informational or warning messages that do not require immediate attention or resolution. They can be sent within a day or even several days after the event.

It is important to understand that alert prioritization is a company-specific process and requires careful analysis and risk assessment. Higher priority alerts should be sent immediately, while less critical ones should be sent at a time convenient for those responsible. [15]

Critical checks and alerts can be enabled in any monitoring system, regardless of the technology used. For example, they can be configured based on monitoring system logs, as well as using various tools and technologies such as performance metrics, event logs, and data analytics. [4]

2.2.3 Efficiency testing options of monitoring solution

In testing of monitoring solution can be used various criteria and indicators to evaluate the effectiveness of controls and warnings as part of the development of a standard set. Here are some possible approaches:

- Accuracy: This is the percentage of correctly identified anomalies, errors, or deviations from expected behavior. Greater accuracy means fewer false positives and more reliable checks and alerts.

- Sensitivity: This is the ability of controls and alerts to detect actual deviations, errors, or anomalies. High sensitivity means that checks and alerts can detect real problems, which can be an important criterion for effective application monitoring in OpenShift. [16]

- Specificity: This is the ability of controls and alerts to avoid false positives and unwarranted alerts. Greater specificity means fewer false positives and more accurate controls and alerts.

- Response Time: This is the time required to respond to checks and alerts for anomalies or error detections. Short response times can be an important indicator of the effectiveness of real-time application monitoring. [17]

- Usability: This measures how easy it is to use and customize controls and alerts. Easier and more intuitive settings help better use OpenShift monitoring solutions. [18]

- Scalability: This is the ability of controls and alerts to work effectively under conditions of high load and application scaling. More scalable solutions can efficiently monitor large and complex applications on OpenShift. [19]

All of these criteria and indicators were used in development to evaluate the effectiveness of a set of standardized checks and alerts, and their compliance with certain requirements and standards can be evaluated using different methods, such as comparison with pre-set thresholds, statistical indicators, peer reviews or real data testing. It is important to select the most appropriate criteria and indicators to suit specific monitoring objectives and requirements in a specific context. [20]

When evaluating criteria and indicators, it is also important to consider their practical applicability and reliability in real conditions. For example, controls and alerts must be flexible enough to adapt to changing application and environmental conditions, and they must be easily customizable and manageable by administrators.

In addition to that various methods and tools can be used to validate the performance of checks and alerts in a monitoring solution. Detailed description of methods which were used to validate the performance of developed monitoring solution. [21]

Using test data and metrics from real-life applications in OpenShift.

Another approach to validate checks and alerts is to use test data and metrics collected from real-life applications in OpenShift. This may include analyzing and comparing monitoring results to expected values (e.g. performance or service availability requirements) and evaluating the effectiveness of controls and alerts in a real-world environment. [22]

For example, historical metrics can be analyzed such as CPU or memory usage, [23] network load, and request duration [13] and compare them to thresholds set in tests. If the monitoring results exceed the set thresholds, the alarm system should trigger and send alerts. Also can be run tests using test data, such as simulating application crashes, shutting down services and manually generating errors, to evaluate the monitoring system's response to such events and the proper functioning of checks and alerts.

Code analysis and review of checks and alerts.

Another method of validating checks and warnings is to analyze and review the code that contains the settings and configurations of the checks and warnings. This can include analyzing Prometheus monitoring rule settings, [24] configuring alerts in Alertmanager, [25] and configuring Grafana panels and dashboards. [26]

By analyzing the code, can be verified that the settings for checks and alerts meet requirements and expectations, that thresholds and re-alert intervals are configured correctly, and that the labels and metrics for collecting metrics and alerts are configured correctly. [13] Also can be reviewed the code for possible bugs, typos, or configuration issues that could cause checks and alerts to work incorrectly.

Regular updates and improvements to controls and alerts.

Checks and warnings are not static elements of a monitoring system and must be regularly updated and improved. As the monitoring system operates, requirements, business rules, application architecture, and other aspects may change, which may require adjustments to controls and alerts.

Regularly updating and improving controls and alerts may include the following steps.

- Analysis of monitoring results and analysis of real usage to identify potential problems or need for optimization.
- Assessing current business rules and requirements and comparing them to the organization's actual goals and strategies. [27]
- Check settings and configuration checks and warnings to identify outdated settings, duplicates, or incorrect settings. [28]
- Identify new or changed metrics, log events, or traces that may be relevant to system monitoring.
- Plan and implement changes to controls and alerts based on identified needs and requirements.
- Testing and validation of updated checks and alerts in test or pre-production environments.

- Documenting and training operational teams and users about changes and new capabilities to the monitoring system. [29]

When changes and updates to controls and alerts are made, it is also necessary to continue to monitor and analyze real-world usage and receive feedback from operations teams and users to evaluate the effectiveness and efficiency of the updated controls and alerts.

Monitoring and analysis of the actual use of controls and alerts.

Once checks and alerts are implemented in a real OpenShift environment, the actual usage of those checks and alerts must be monitored and analyzed. Actual usage monitoring allows to observe the operation and effectiveness of controls and warnings in real-world conditions, as well as to identify potential problems or the need for improvement. [30]

Various monitoring tools such as Prometheus, Grafana, Alertmanager, as well as log analyzers, monitoring systems and other tools for collecting and analyzing system performance and status data can be used for this purpose.

Real-world usage analysis can include monitoring metrics and alerts, analyzing logs and traces, [31] as well as analyzing real-time and historical system performance and availability data. This allows not only detect technical problems such as crashes or incorrect settings, but also analyze the effectiveness of alerts, their relevance, timeliness and accuracy.

Feedback from operational teams and users.

Another important approach to validating controls and alerts is to get feedback from workgroups and users. They can be a valuable source of information about how controls and warnings work effectively in real-world situations, and can identify potential problems or suggest improvements. [32]

Various channels can be used to collect feedback, such as feedback from users, conducting interviews with operational teams, organizing joint meetings or discussions. This helps collect real feedback and opinions about how controls and alerts work, and make changes to settings or configurations based on that feedback. [32]

2.2.4 Tools

A combination of tools such as Prometheus, Grafana and Alertmanager were chosen for this project. This set of tools allows to collect and analyze metrics, create graphical dashboards, and send notifications about system anomalies. With this range of tools, the customer can quickly respond to issues and minimize application downtime, as well as improve the overall reliability and performance of the OpenShift infrastructure.

Prometheus is an open source monitoring system used to collect and store time series of metrics from various data sources. It provides insight into the performance of applications, containers, and servers, as well as a powerful query language and flexible visualization tools that allow quickly and easily monitor the performance of the entire system. Prometheus has built-in capabilities for logging and collecting metrics, and it also allows to use exporters to collect metrics from other applications. It is the primary tool for monitoring container applications in OpenShift. [33]

Grafana is a data visualization system that allows to create informative graphs and dashboards based on the data collected by Prometheus. This allows to monitor key metrics in real time and quickly react to changes in the system. Grafana also offers the ability to create notifications based on certain conditions and set thresholds for quick notification of system anomalies. [34]

Alertmanager is an alert management system that integrates with Prometheus to automatically notify of system-related issues. It allows to define alert routing rules, customize notifications for different types of events, and integrate with third-party tools to quickly respond to issues. Alertmanager allows timely and accurately inform the people responsible for system maintenance about problems. [35]

One of the advantages of the Prometheus, Alertmanager and Grafana suite is their flexibility and ability to adapt to specific requirements and monitoring needs in an OpenShift environment. Prometheus offers powerful monitoring capabilities with automatic detection of monitoring objects and flexible alert rules. [6] In turn, Alertmanager offers flexible alert rule configuration and alert management, including alert grouping and suppression. Grafana offers data visualization capabilities with support for flexible customization of dashboards and panels.

Another advantage of Prometheus, Alertmanager and Grafana is their ecosystem of plugins and integrations. Prometheus has many exporters that allow to collect metrics from different systems such as databases, web servers, container clusters, and more. [36] Alertmanager also has support for different alert channels, allowing to integrate with different alerting tools. Grafana, in turn, has many plugins and integrations that allow to visualize data from different data sources. [37]

In thesis is also used Kustomize to automate the deployment and configuration of monitoring tools in OpenShift. Kustomize is a tool that allows to configure Kubernetes applications using YAML configuration files. This allows large and complex applications to be broken down into smaller, more manageable parts, making application deployment and management easier.

Kustomize also integrates with Git, making it easy and efficient to manage configuration changes and automatically update application and monitoring tools. This greatly simplifies the application deployment and management process and ensures faster and more efficient work. [11]

3 Development

Before starting the development of the monitoring solution, a scheme of the future project was developed. Figure 1 shows this scheme. This demonstrates the idea of developing the Prometheus, Grafana and Alertmanager tools separately. Then, after each tool is created, a separate directory is created containing all previously developed tools and their updated configurations.

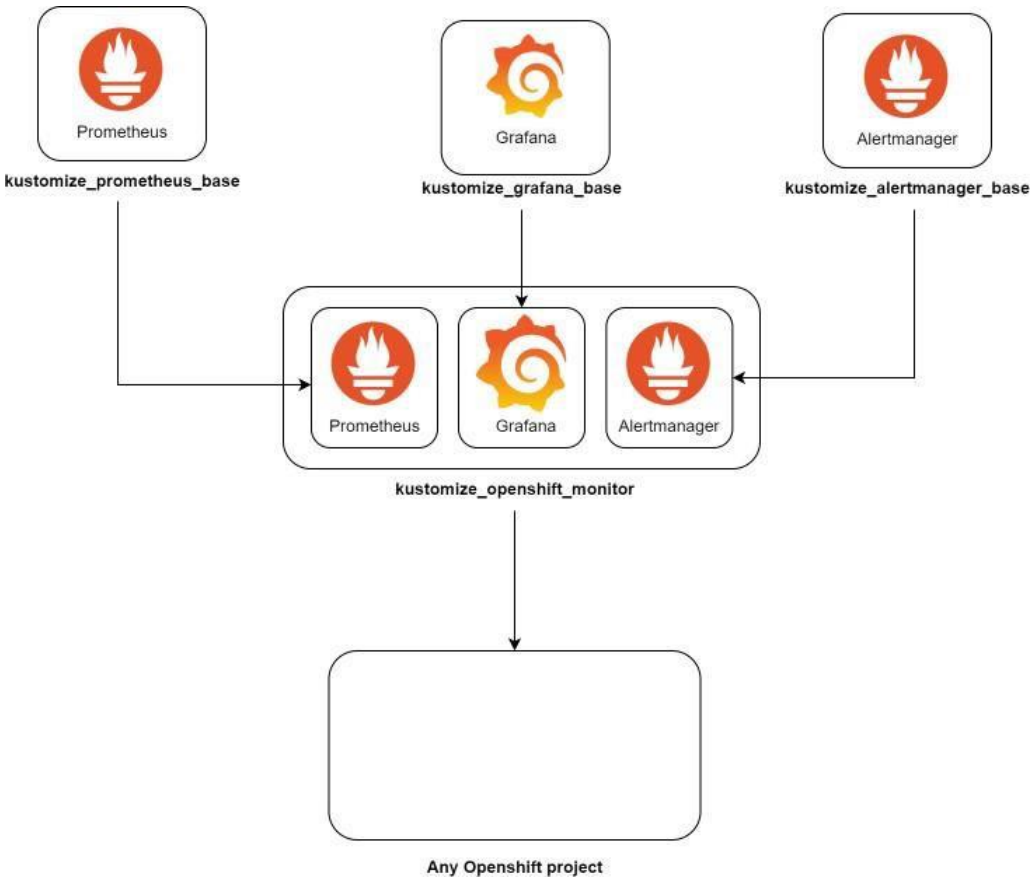

Figure 1. Scheme of OpenShift monitoring project

Source: author's scheme

This solution allows to integrate each tool individually if needed, or to integrate all at once using a common directory with all configurations and tools. This gives the flexibility to customize monitoring solution according to project requirements.

It is also worth noting that each tool can be used independently of the others, making it easy to replace or update them without affecting other components of the monitoring solution. But using all the tools together gives a maximum monitoring efficiency, making it easy to capture and analyze application performance and availability data.

3.1 Grafana development

It was decided to start the development of the monitoring solution with Grafana, as the development of this tool for virtualization had its own subtleties, which complicated the process in its own way. Some applications in the company for which this project is being done work on the OpenShift 3.x version platform and have not been migrated yet to the OpenShift 4.x version. In the OpenShift 3.x version, unlike the newer versions of OpenShift 4.x version, no special application used to run containerization applications in the cluster init-container. This, in turn, in addition to creating a Grafana-init container, requires the creation of an additional so-called Grafana-init container. In general, for the development of graphs on the OpenShift platform, two containers will need to be launched.

The first stage of development was the creation of the kustomizaon.yaml file, as it was decided to use Kustomize as a tool for configuring plain and free YAML files. This tool helps to quickly and easily prepare YAML files for deployment in Kubernetes. The config kustomizaon.yaml file can be seen in the Appendix 2.

After this stage, the development of the resource directory began, which includes all the necessary YAML configurations for the OpenShift application.

It took five different configurations to run the Grafana container:

- grafana.deploymentconfig.yaml - This program file configures the Grafana deployment to OpenShift with a DeploymentConfig object declaration. It defines deployment parameters, such as the number of replicas, image usage, environment settings, and other parameters necessary for launching and scaling a Grafana instance.

- grafana.imagestream.yaml - This file defines the ImageStream configuration, which represents an abstraction of the container image in OpenShift. ImageStream

allows to manage container images and provides automatic update of images in the application when creating new versions of images.

- grafana.route.yaml - This file is for configuring the route to Grafana from outside the OpenShift cluster. Route allows to create an external URL address that provides safe and accessible external access to service applications within the cluster.

- grafana.service.yaml - This file defines the configuration of the Grafana service (Service), which provides network access to the application within the cluster. The service allows other OpenShift objects to interact with Grafana over the network within the cluster.

- grafana.buildconfig.yaml - This BuildConfig process file, which defines the process of building a grafana image. BuildConfig allows to automate the process of building and updating the Grafana image, including automatic building when changing the source code or updating the base image.

After writing these configurations, it is necessary to create a grafana.env file, which is necessary for the full operation of the graph. Various environments were specified in this file, as well as the port on which the application works, various paths of the graph, plug-ins and the dashboard itself, as well as many other things. File with full configuration can be seen in Appendix 3.

When everything was set up, a trial run of the Grafana container was made to check its functionality. Start up was carried out by oc apply –k . command.

Figure 2 shows that the Grafana was successfully launched, but still without a dashboard. To install a dashboard in Grafana, was needed to launch an init container, which will be done below.

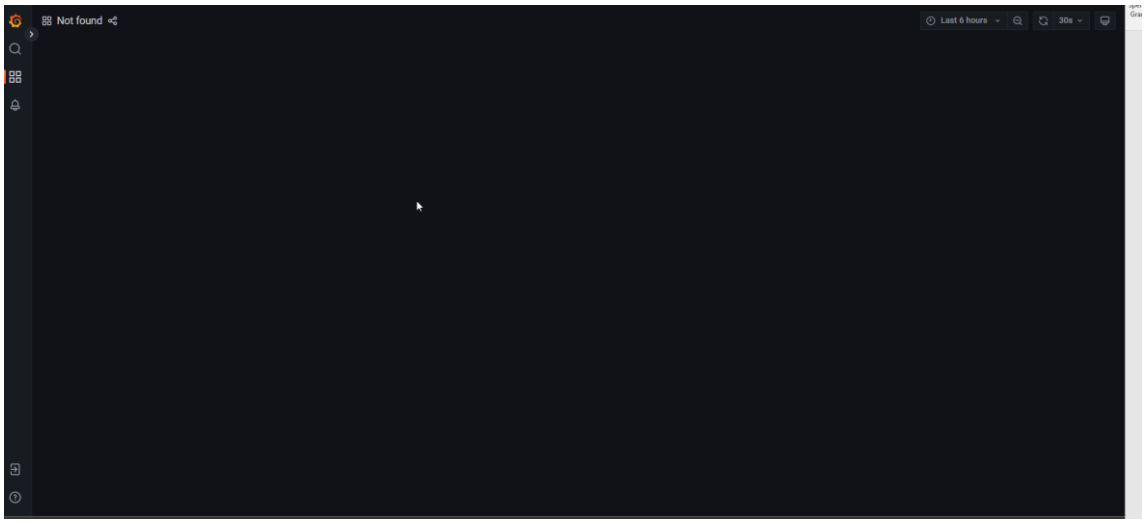

Figure 2. Running Grafana

Source: author's screenshot

After installing Grafana and checking its performance, it's time to develop an init container. To do this, the author had some time to think about the implementation of this stage. The idea was to launch a separate init container using the Docker file. The job of the init container is to insist on Grafana configuration files. This includes things like dashboards, ldap, plugins, and datasources.

For these purposes, a Dockerfile and a src folder were created, in which all the necessary Grafana configuration files were placed. Dockerfile configurations can be found in Appendix 4.

After all the operations done, it was necessary to create two configuration files in the "resource" directory, which was already discussed earlier. These files are grafana-init.imagestream.yaml and grafana-init.buildconfig.yaml. Grafana-init.buildconfig.yaml is needed in turn to run the Dockerfile, and grafana-init.imagestream.yaml is needed to automatically update the container init image.

Structure of Grafana directory:

```
.
├── base
│   ├── configs
│   │   └── grafana.env
│   ├── dockerfiles
│   │   ├── grafana
│   │   │   └── Dockerfile
│   │   └── initcontainer
│   │       ├── Dockerfile
│   │       └── src
│   │           ├── dashboards
│   │           └── provisioning
│   │               ├── dashboards
│   │               ├── datasources
│   │               ├── ldap.toml
│   │               ├── notifiers
│   │               └── plugins
│   │                   └── default.yaml
│   ├── kustomization.yaml
│   └── resources
│       ├── grafana.buildconfig.yaml
│       ├── grafana.deploymentconfig.yaml
│       ├── grafana.imagestream.yaml
│       ├── grafana-init.buildconfig.yaml
│       ├── grafana-init.imagestream.yaml
│       ├── grafana.route.yaml
│       └── grafana.service.yaml
└── README.md
```

After all the operations done, Grafana was redeployed in the OpenShift platform, but first of all was needed to login to OpenShift project. It was done by oc login -u=your.user.name --server=https://api.******:6443 command. When user was logged in to the right project was used oc apply k . command to deploy changes.

Within a minute of running the command, the Pod with the Grafana and Grafana init containers were successfully launched. The result of a successful launch is shown in the Figure 3.

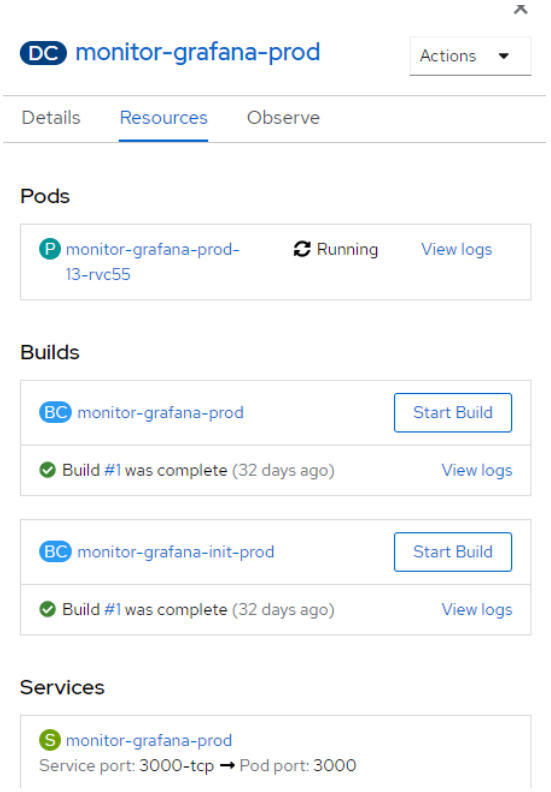

Figure 3. Grafana pod in OpenShift

Source: author's screenshot

After deploying Grafana in the OpenShift environment, the next step is to connect it to the Prometheus data source and create a dashboard based on the received metrics. This will be done in paragraph 3.4.

3.2 Prometheus development

After successfully launching Grafana, it was necessary to deploy Prometheus on OpenShift. This is necessary so that it is possible to receive metrics from the OpenShift environment and send notifications based on them. Grafana also uses Prometheus as a data source to use Prometheus metrics in OpenShift to illustrate cluster health.

The Prometheus catalogue turned out to be significantly smaller than that of Grafana (see paragraph 3.1). To deploy Prometheus in OpenShift, it was only necessary to create the configuration file kustomization.yaml and the resources folder with all the configuration files.

List of YAML configurations files:

- prometheus-configmap.yaml
- prometheus-rollbinding.yaml
- prometheus-route.yaml
- prometheus-service.yaml
- prometheus-serviceaccount.yaml
- prometheus-statefulset.yaml

The main task of working with Prometheus was to understand which containers would be needed for Prometheus to work fully. After analysis, it was decided to run the following list of containers to deploy Prometheus:

- Prometheus Container: this container contains the Prometheus application itself, which is responsible for collecting, storing, and processing system, application, or service health metrics. A Prometheus container can also contain settings such as metrics collection intervals, alert rules, and other options that determine its behaviour.

- Prometheus ConfigMap Reload Container: this container is responsible for dynamically reloading the Prometheus configuration from the ConfigMap. A ConfigMap is a Kubernetes resource that contains application configuration, such as target service addresses for monitoring or alerting rules. The Prometheus ConfigMap Reloader container listens for ConfigMap changes and automatically reloads the Prometheus configuration so that the changes take effect without restarting the entire Prometheus container.

- kube-state-metrics container: this container is responsible for collecting metrics about the state of Kubernetes objects such as folders, services, replicas, and others. kube-state-metrics collects information about the state of Kubernetes resources and provides it in a format that Prometheus understands, allowing to use these metrics to monitor and analyze the state of Kubernetes cluster.

Together, these containers allow to configure and deploy Prometheus monitoring on Kubernetes (OpenShift), collecting metrics about system health and Kubernetes resources, as well as setting alert rules to quickly respond to issues and keep the system stable. All of this is configured in the prometheus-statefulset.yaml file.

After setting up the prometheus-statefulset.yaml configurations, another important task was to set up the prometheus.yml and prometheus-rules correctly. These two configurations are for setting up metrics collection as well as creating conditions for sending a notification. ConfigMap was used for this. A ConfigMap, in turn, is a resource used to store configuration data such as application settings, environment settings, secrets and other settings that may change during application execution. In this case, it was used to configure Prometheus without the need to create separate prometheus.yml and prometheus-rules files, giving the application the flexibility to customize.

Structure of Prometheus directory:

```
.
├── base
│   ├── kustomization.yaml
│   └── resources
│       ├── prometheus-configmap.yaml
│       ├── prometheus-route.yaml
│       ├── prometheus-service.yaml
│       ├── prometheus-rolebinding.yaml
│       ├── prometheus-serviceaccount.yaml
│       └── prometheus-statefulset.yaml
└── README.md
```

After all these configurations, the Prometheus pod was started with the containers, Prometheus, prometheus-configmap-reload and kube-state-metric. The launch was successful, which can be seen in Figure 4.

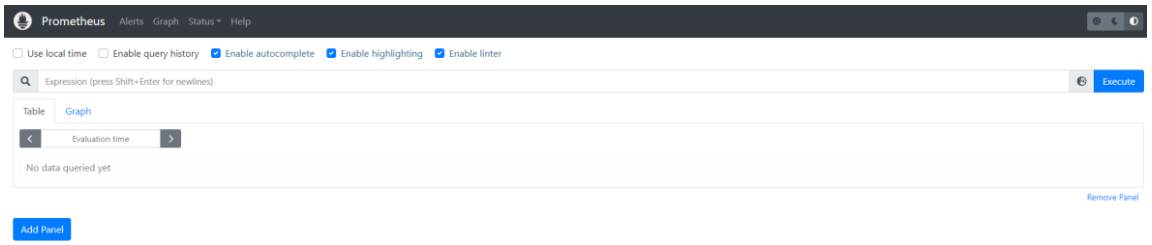

Figure 4. Running Prometheus

Source: author's screenshot

3.3 Alertmanager development

Since the installation and configuration of Grafana and Prometheus were successfully completed, the next stage was the development of Alertmanager. Alertmanager is necessary here for receiving alerts from Prometheus and subsequent alerts. When choosing a way to send alerts, it was decided to use sending emails, as it is the most optimal solution.

For the development of Alertmanager, such a structure was used as in the situation with Prometheus. A custom file was created that specified the path to other configurations, such as alertmanager-deployment.yaml, alertmanager-service.yaml and alertmanager-route.yaml. Besides starting the container with the application, Alertmanager requires starting the container with configmap-reload, which performs the same function as in the case of Prometheus (described in paragraph 3.2). The only difference between deploying Prometheus and Alertmanager, besides the image itself, was the setting of the application's configuration. In the case of Prometheus, the file prometheus-configmap.yaml was created, in which the configurations for the files prometheus-rules and prometheus.yml were specified. Things are a little different with Alertmanager. It was decided to use ConfigMapGenerator, which allows to specify the path to a file with a configuration application. This method does not fit with Prometheus, as it requires the setting of two configurations at once, and it is impossible with ConfigMapGenerator.

After the kustomization.yaml file was successfully created, the resource path and Alertmanager configuration were specified, it was necessary to create an Alertmanager configuration. At this stage of development, a default configuration was created without any special receivers. This was connected with the fact that in the next paragraph 3.4 a

separate directory will be created, which will include the already updated configuration file of Alert manager, Prometheus and Grafana. This structure was made so that, if necessary, it was possible to run Prometheus, Alertmanager and Grafana separately, or in the case of monitoring OpenShift clusters - all together what was described in paragraph 3.

Structure of Alertmanager directory:

```
.
├── base
│   ├── configs
│   │   └── config.yml
│   ├── kustomization.yaml
│   └── resources
│       ├── alertmanager-deployment.yaml
│       ├── alertmanager-route.yaml
│       └── alertmanager-service.yaml
└── README.md
```

The last step of this paragraph was starting the Alertmanager, which was also executed with the oc apply k . command. After a minute of waiting, the Pod with Alertmanager was successfully launched, and by going to the Alertmanager link, was able to verify the functionality of the application (see Figure 5).

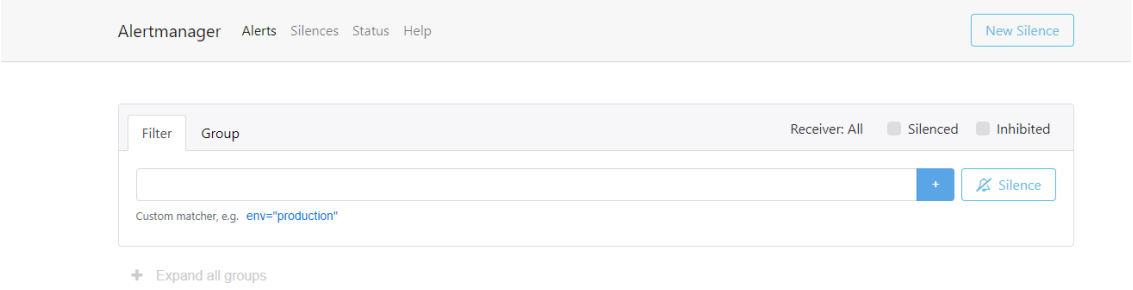

Figure 5. Running Alertmanager

Source: author's screenshot

3.4 Developing the OpenShift monitoring directory

All the necessary tools for monitoring have been developed, so the last step was to create a common directory that includes all the tools previously deployed in the OpenShift platform (Prometheus, Grafana, Alertmanager), as well as all the necessary updated

configuration files for them. One of the objectives of this approach is the maximum ease of implementation of such a monitoring solution in different OpenShift projects. This is due to the fact that all the necessary tools will be launched and configured using one directory, which greatly simplifies its use in the future.

First of all, as with the deployment of Prometheus, Alertmanager and Grafana, a kustomization.yaml file was created. It contained links to previously created tools, as well as files with new configurations for Prometheus and Alertmanager (see Appendix 5).

Next, the configuration files themselves were created. A separate config file has been created for Alertmanager since it uses ConfigMapGenerator (was specified in paragraph 3.3). In the config file, a receiver, an alertgroup, a ripple interval and much more were configured. The complete file can be found in Appendix 6.

In the case of Prometheus, a separate prometheus-config-patch.yaml patch was created, which in turn changes the Promeheus configuration files.

It indicated those metrics that, in the opinion of the author and the customer, would be the most effective at this stage of development. The list of metrics and their values was indicated in Table 1.

Table 1. Prometheus rules alert metrics

	Metric	Meaning
1	kube_pod_status_phase{phase="Failed"} == 1	Check failed pods
2	sum by(pod)((kube_pod_status_phase{phase="Running"} == 1) * on(pod) group_right() kube_pod_container_status_restarts_total) > 1	Check amount of restarts in pods
3	(100 - (sum(kube_pod_container_resource_requests{resource="memory"}) by (pod) / sum(kube_pod_container_resource_limits{resource="memory"}) by (pod) * 100) < 10) * on(pod) group_right(namespace) kube_pod_info	Check available memory in container
4	(100 - (sum(kube_pod_container_resource_requests{resource="cpu"}) by (pod) / sum(kube_pod_container_resource_limits{resource="cpu"}) by (pod) * 100) < 10) * on(pod) group_right(namespace) kube_pod_info	Check available CPU in container
5	kube_statefulset_status_replicas_ready < kube_statefulset_replicas	Checks

		not ready replicas

After updating the configuration files for Alertmanager and Prometheus, the last task was to properly configure Grafana, as well as create a dashboard for monitoring applications in OpenShift based on the metrics received from Prometheus.

To do this, it was necessary to copy the previously created Grafana directory and place it in the OpenShift monitoring directory. It also required two key changes:

- edit the datasource file - it is necessary to add a link to the previously created Prometheus there so that Grafana can use it as a datasource and collect metrics from it;
- create a dashboard - necessary in order to visualize the necessary data that will be required to identify problems with applications in the OpenShift environment.

In the Table 2 can be found metrics that were used in the Grafana dashboard.

Table 2. Grafana dashboard used metrics

	Metric	Meaning
1	count(kube_pod_status_phase{phase="Running"} == 1)	Shows amount of running pods
2	count(kube_pod_status_phase{phase="Running"} == 1) - count((kube_pod_status_phase{phase="Running"} == 1) * on (pod) group_right() kube_pod_status_ready {condition="true"} == 1)	Shows amount of no ready pods
3	sum(kube_pod_status_phase{phase="Failed"})	Shows amount of failed pods
4	(kube_pod_status_phase{phase="Running"} == 1) * on (pod) group_right() kube_pod_status_ready {condition="true"}	Shows amount of ready pods
5	sum by(pod)((kube_pod_status_phase{phase="Running"} == 1) * on(pod) group_right() kube_pod_container_status_restarts_total)	Shows amount of restarts in pods
6	100 - (sum(kube_pod_container_resource_requests{resource="memory"}) / sum(kube_pod_container_resource_limits{resource="memory"}) * 100)	Shows available

		memory in container
7	100 - (sum(kube_pod_container_resource_requests{resource="cpu"}) / sum(kube_pod_container_resource_limits{resource="cpu"}) * 100)	Shows available CPU in container
8	kube_pod_restart_policy{pod!~".*deploy", pod!~".*build"}	Shows pod Restart Policy
9	kube_statefulset_status_replicas_ready - kube_statefulset_replicas	Shows not ready replicas

OpenShift monitor directory structure:

```
.
├── base
│   ├── alertmanager
│   │   └── configs
│   │       └── config.yml
│   ├── Grafana
│   │   └── dockerfiles
│   │       └── initcontainer
│   │           ├── Dockerfile
│   │           └── src
│   │               ├── dashboards
│   │               └── provisioning
│   │                   ├── dashboards
│   │                   │   └── all.yml
│   │                   └── datasources
│   │                       └── datasource.yml
│   └── prometheus
│       └── patches
│           └── prometheus-config-patch.yaml
└── README.md
```

After all the configurations, the last step is to implement the OpenShift monitor directory into the OpenShift project, which will be done in paragraph 3.5.

3.5 Implementation of OpenShift Monitoring into OpenShift Working Project

The last step in the development of a Standardized Set of Checks for Apps in OpenShift is to implement it into a ready-made project with working apps.

To do this, it was necessary to create a new folder with a kustomization.yaml file and the Grafana repository in the directory with already running applications. In the kustomization file, the previously created OpenShift monitor project was specified in the resources, as well as the Grafana folder. The kustomization.yaml file can be found in Appendix 7.

In the Grafana directory, a repository was created with the updated Grafana.env, in which the path to the Grafana dashboard was specified, as well as the project's OpenShift namespace. The namespace was specified because it is used in the name of the Prometheus datasource, which in turn improves the flexibility of implementing OpenShift monitoring into any project.

Also, another kustomization.yaml file was created in the Grafana directory, which contained a link to the init container and to the previously created kustomize_grafana_base directory. It also included configMapGenerator with grafana.env. The kustomization.yaml file can be found in Appendix 8.

The structure of the implemented OpenShift monitoring in the OpenShift working project:

```
.
├── services
    ├── license-statistics-backend
    │   └── kustomization.yaml
    └── OpenShift_monitor
        ├── Grafana
        │   ├── configs
        │   │   └── Grafana.env
        │   └── kustomization.yaml
        └── kustomization.yaml
```

The last step was the launch of the previously developed and configured OpenShift monitoring in the working project. After running OpenShift Monitoring, Prometheus (see Figure 6), Grafana (see Figure 7), Alertmanager (see Figure 8) were automatically successfully deployed in the selected OpenShift working project.

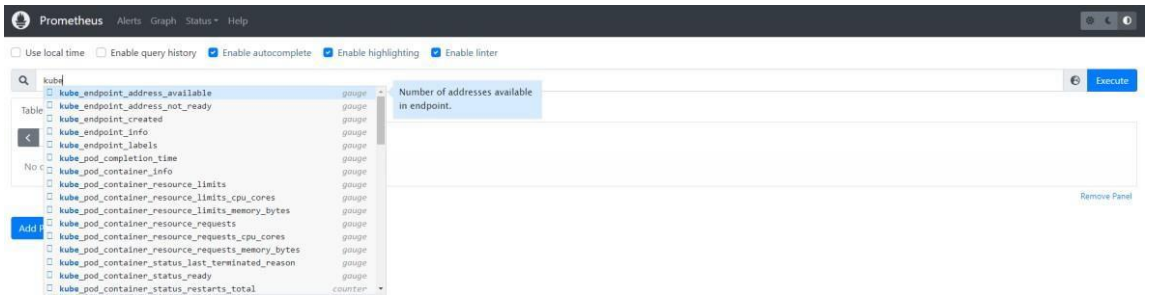

Figure 6. Running Prometheus

Source: author's screenshot

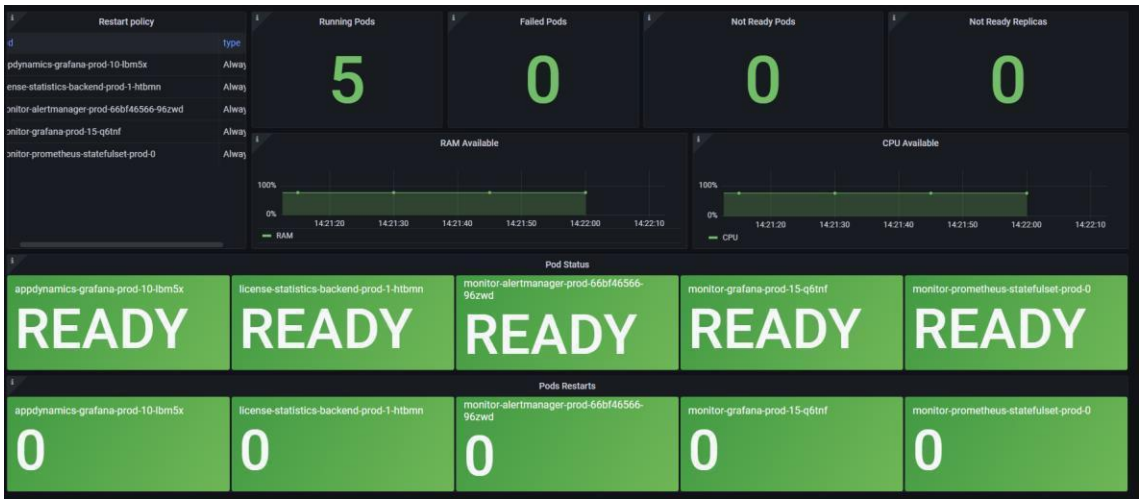

Figure 7. Running Grafana

Source: author's screenshot

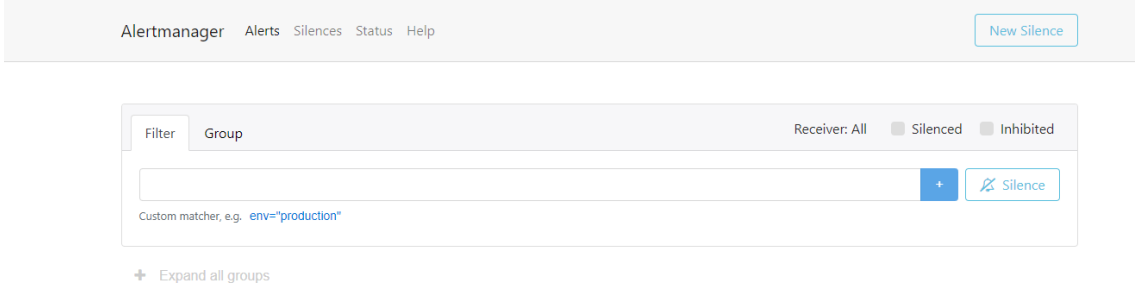

Figure 8. Running Alertmanager

Source: author's screenshot

4 Results, conclusion and future plans

4.1 Testing results

On the basis of paragraph 2.2.3, a number of tests were carried out to verify the effectiveness of this monitoring solution.

4.1.1 Using test data and metrics from real-life applications in OpenShift.

Test metrics were created based on real-world application data to verify proper operation of monitoring and alerts. This allowed us to test the performance of monitoring and alerts in real-world use cases and make sure they were configured correctly.

As a result of the check, it was noted that the notifications were successfully sent by post (see Figure 9). This indicates that monitoring and alerts are configured correctly and are working correctly. This approach helps to avoid possible problems and errors in the application, ensuring their more reliable and efficient operation.

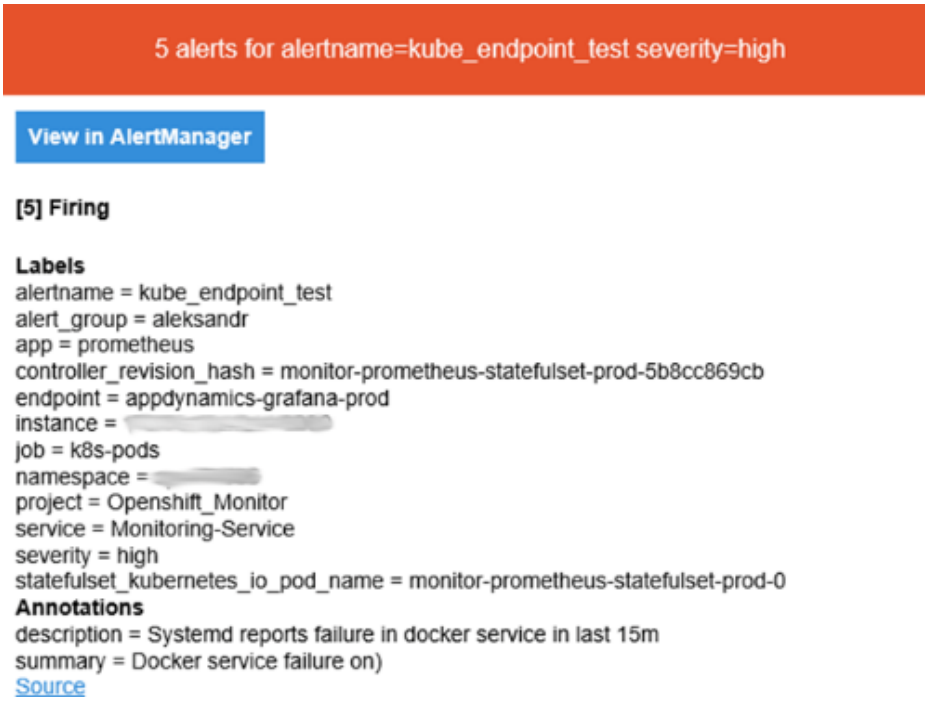

Figure 9. Notification from Alertmanager

Source: author's screenshot

4.1.2 Code analysis and review of checks and alerts.

As part of the testing of the monitoring solution, code analysis and verification of controls and warning messages were performed. To do this, the code with changes was uploaded to the repository and a separate pull request was created. After the development team analyzed the code, they confirmed its correctness and reliability, and also confirmed the correctness of the writing of checks and warnings and the entire monitoring structure of the solution in total (see Figure 10).

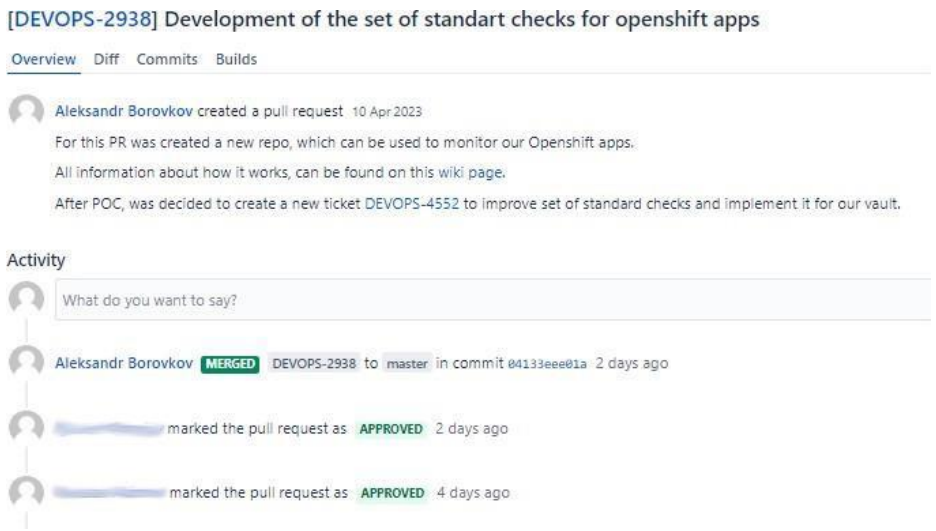

Figure 10. Code review by members of the team

Source: author's screenshot

In this case, code verification allowed the author to verify the correctness of the compiled code. This confirms that the process of code review and verification of checks and warnings is very important in the application development process and enables application reliability.

4.1.3 Regular updates and improvements to controls and alerts.

The author added new metrics to solution tracking as needed. He regularly monitored the performance of the application and analyzed the monitoring data to determine where control and alerts could be improved.

The addition of new metrics allowed for better control and alerts on solution tracking and ensured that the application as a whole worked more reliably and efficiently. This helped to respond quickly to potential problems and minimize potential risks.

4.1.4 Monitoring and analysis of the actual use of controls and alerts

As part of checking the reliability and stability of the application, the author conducted an intentional crash test, where the application was deliberately "broken" to test its response and performance in unforeseen situations. To verify the application and its stability, the author monitored the changes in the Grafana monitoring map.

During intentional crash testing, the app was found to not work and was constantly reloading. Information obtained through Grafana confirmed these observations. The crash test results can be seen in Figure 11.

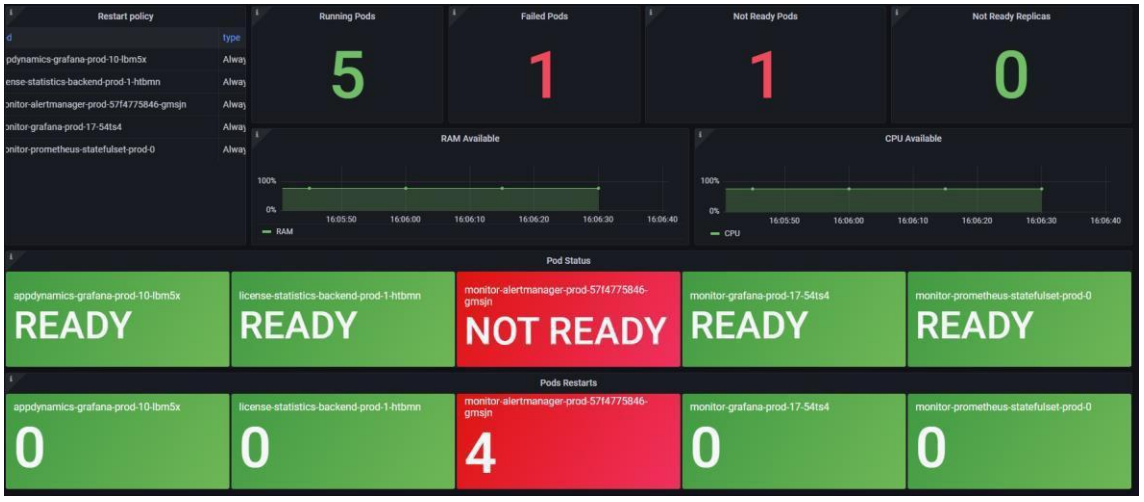

Figure 11. Grafana dashboard after crashing the application

Source: author's screenshot

The testing was successful and helped identify the problem in the application. Therefore, performing intentional crash testing and monitoring with Grafana is an important step in ensuring application reliability and stability. This allows to identify and correct problems in the operation of the application, which ensures a more efficient and secure operation of the application as a whole.

4.1.5 Feedback from operational teams and users.

As part of the "Feedback from operational teams and users" section, the author received feedback from operational teams and users about the solution monitoring work. After receiving feedback, the author took into account all comments and opinions and made the necessary corrections and additions while tracking the solution.

Adjustments and improvements made based on feedback have improved the tracking of the solution and eliminated identified shortcomings. This ensured more reliable and efficient operation of the application as a whole.

4.2 Conclusion

The final result of this thesis confirms the effectiveness and importance of using monitoring in modern systems, which allows to ensure the stable and safe operation of applications. In the course of the research, a monitoring system based on Prometheus and Grafana was developed and implemented, which allows timely identification and correction of possible problems in the operation of applications.

The conducted tests and the analysis of the results proved that the monitoring system works properly and allows timely response to possible problems. In addition, it was found that regular updating and improvement of control mechanisms and alerts can improve the reliability and efficiency of the system.

Although progress has been made, several areas for improvement have been identified during the work. For example, the monitoring dashboard can be enhanced with new charts to improve visibility and identify potential issues faster. It is also important to add new metrics to improve tracking accuracy and improve alerts. Perhaps must be consider setting up alerts not only for email but also for other communication channels such as Slack to respond to issues more quickly and efficiently. These improvements further improve the efficiency and reliability of the monitoring solution.

4.3 Future plans

In particular, many new metrics are planned to be added to Prometheus in the future, including useful exporters, in addition to kube-state-metrics. There are also plans to improve the dashboard and add many new reporting metrics.

For Grafana, there are plans to add new panels to display application performance data, use graphs to show the dynamics of metrics, and add the ability to configure data grouping and filtering in dashboards.

For Alertmanager, there are plans to add new metrics to report on issues in applications, use a grouping mechanism to combine multiple alerts, and integrate with event management systems. Additionally, might be considered using an auto-scaling mechanism to quickly respond to changes in load and ensure higher application availability.

For more efficient monitoring and management of applications on the OpenShift platform, integration with other monitoring systems such as Zabbix or Nagios can also be considered. Knowledge sharing between development and management teams is also an important aspect of improving application monitoring on the OpenShift platform.

In general, improving application monitoring on the OpenShift platform requires an integrated approach in the created monitoring solution and continuous development and improvement of monitoring and alerting tools, as well as cooperation and knowledge sharing between development and admin teams.

5 Summary

In this thesis, a monitoring solution based on a combination of Prometheus, Alertmanager and Grafana tools was developed and implemented, which ensures the stable operation of applications on the OpenShift platform.

A series of tests were conducted using real data and application metrics to verify the health and effectiveness of monitoring. The audit showed that the monitoring system works correctly and allows timely response to possible problems.

Although the work performed was considered successful by the author and the client, several areas of improvement were found. In particular, the dashboard can be improved by adding new graphs to improve visibility and quickly identify potential issues. In addition, it is important to add new metrics to improve tracking accuracy and better alerts. Also was considered setting up alerts not only for email, but also for other communication channels such as Slack to respond to issues more quickly and efficiently.

Intentional crash tests were conducted where the application was intentionally "broken" to test its response and performance in unexpected situations. The audit showed that the use of Grafana's failover and monitoring mechanism is an important step to ensure the reliability and stability of the application. This allows to quickly identify and fix problems with app, making it more efficient and secure overall.

The experience gained made it possible to identify several directions for the further development of the monitoring solution. In particular, many new metrics are planned to be added to Prometheus, including useful exporters other than the cube-state metric.

References

[1] Red Hat, “Red Hat OpenShift enterprise Kubernetes container platform.” https://www.redhat.com/en/technologies/cloud-computing/OpenShift (accessed Feb. 28, 2023).

[2] Docker, “What is a Container? | Docker,” Nov. 11, 2021. https://www.docker.com/resources/what-container/ (accessed Mar. 02, 2023).

[3] Kubernetes, “Overview,” *Kubernetes*. https://kubernetes.io/docs/concepts/overview/ (accessed Mar. 04, 2023).

[4] Prometheus, “Metric and label naming | Prometheus.” https://prometheus.io/docs/practices/naming/ (accessed Mar. 10, 2023).

[5] Red Hat, “About Logging | Logging | OpenShift Container Platform 4.9.” https://docs.OpenShift.com/container-platform/4.9/logging/cluster-logging.html (accessed Mar. 20, 2023).

[6] “Monitoring overview | Monitoring | OpenShift Container Platform 4.11.” https://docs.OpenShift.com/container-platform/4.11/monitoring/monitoring-overview.html (accessed Mar. 01, 2023).

[7] Prometheus, “Alerting overview | Prometheus.” https://prometheus.io/docs/alerting/latest/overview/ (accessed Mar. 15, 2023).

[8] Red Hat, “Ansible is Simple IT Automation.” https://www.ansible.com (accessed Mar. 11, 2023).

[9] Puppet, “Puppet Infrastructure & IT Automation at Scale | Puppet by Perforce.” https://www.puppet.com/ (accessed Mar. 01, 2023).

[10] Chef, “Chef Software DevOps Automation Solutions | Chef.” https://www.chef.io/ (accessed Mar. 11, 2023).

[11] Kustomize, “Kustomize - Kubernetes native configuration management.” https://kustomize.io/ (accessed Mar. 05, 2023).

[12] Planview, “What Is Resource Management and Why Is It Important?,” *Planview*. https://www.planview.com/resources/guide/resource-management-software/resource-management-leverage-people-budgets/ (accessed Mar. 12, 2023).

[13] Datadog, “Monitoring 101: Collecting the right data,” *Monitoring 101: Collecting the right data*. https://www.datadoghq.com/blog/monitoring-101-collecting-data/ (accessed Feb. 28, 2023).

[14] Red Hat, “Managing alerts | Monitoring | OpenShift Container Platform 4.11.” https://docs.OpenShift.com/container-platform/4.11/monitoring/managing-alerts.html (accessed Feb. 27, 2023).

[15] bwren, “Azure Monitor best practices: Alerts and automated actions - Azure Monitor,” Mar. 10, 2023. https://learn.microsoft.com/en-us/azure/azure-monitor/best-practices-alerts (accessed Mar. 02, 2023).

[16] A. Mironov and P. Doronkin, “An Analysis of Sensitivity of the Monitoring System of Helicopters to Faults of their Blades,” *Mechanics of Composite Materials*, May 01, 2021. https://doi.org/10.1007/s11029-021-09948-z (accessed Mar. 15, 2023).

[17] Netdata, “Response Time Monitoring | Netdata.” https://www.netdata.cloud/response-time-monitoring/ (accessed Mar. 01, 2023).

[18] usability, “Usability Evaluation Basics,” Oct. 08, 2013. https://www.usability.gov/what-and-why/usability-evaluation.html (accessed Mar. 03, 2023).

[19] PageWriter-MSFT, “Monitor performance for scalability and reliability - Microsoft Azure Well-Architected Framework,” Nov. 30, 2022. https://learn.microsoft.com/en-us/azure/well-architected/scalability/monitor-scalability-reliability (accessed Mar. 06, 2023).
[20] sopact, “Effective Monitoring and Evaluation for Impactful Results.” https://www.sopact.com/monitoring-and-evaluation (accessed Feb. 27, 2023).
[21] ninjaOne, “Server Monitoring and Alerting Software: Best Practices | NinjaOne,” Mar. 07, 2022. https://www.ninjaone.com/blog/server-monitoring-and-alerting/ (accessed Mar. 11, 2023).
[22] evalcomminity, “What is the difference between monitoring and evaluation? EvalCommunity,” Apr. 05, 2023. https://www.evalcommunity.com/career-center/what-is-the-difference-between-monitoring-and-evaluation/ (accessed Feb. 27, 2023).
[23] Prometheus, “Metric types | Prometheus.” https://prometheus.io/docs/concepts/metric_types/#gauge (accessed Mar. 09, 2023).
[24] Prometheus, “Alerting rules | Prometheus.” https://prometheus.io/docs/prometheus/latest/configuration/alerting_rules/ (accessed Mar. 08, 2023).
[25] Prometheus, “Configuration | Prometheus.” https://prometheus.io/docs/alerting/latest/configuration/ (accessed Mar. 10, 2023).
[26] Grafana, “Panels and visualizations | Grafana documentation,” *Grafana Labs*. https://grafana.com/docs/grafana/latest/panels-visualizations/ (accessed Mar. 01, 2023).
[27] Bain, “Balanced Scorecard,” *Bain*, Jan. 31, 2023. https://www.bain.com/insights/management-tools-balanced-scorecard/ (accessed Mar. 03, 2023).
[28] J. Demian, “10 Best Synthetic Monitoring & Testing Tools [2023 Comparison],” *Sematext*, Jan. 04, 2023. https://sematext.com/blog/synthetic-monitoring-tools/ (accessed Mar. 21, 2023).
[29] altexsoft, “Technical Documentation in Software Development: Types, Best Practices, and Tools,” *AltexSoft*. https://www.altexsoft.com/blog/business/technical-documentation-in-software-development-types-best-practices-and-tools/ (accessed Mar. 08, 2023).
[30] S. Cooper, “Application Monitoring - Best Practices,” *Comparitech*, Sep. 09, 2022. https://www.comparitech.com/net-admin/application-monitoring-best-practices/ (accessed Mar. 04, 2023).
[31] techtarget, “The 3 pillars of observability: Logs, metrics and traces | TechTarget,” *IT Operations*. https://www.techtarget.com/searchitoperations/tip/The-3-pillars-of-observability-Logs-metrics-and-traces (accessed Mar. 13, 2023).
[32] actiTime, “The Power of Feedback: How to Use It to Grow and Improve,” Apr. 23, 2021. https://www.actitime.com/project-management/importance-of-feedback (accessed Mar. 07, 2023).
[33] Prometheus, “Overview | Prometheus.” https://prometheus.io/docs/introduction/overview/ (accessed Mar. 02, 2023).
[34] Red hat, “What is Grafana?” https://www.redhat.com/en/topics/data-services/what-is-grafana (accessed Feb. 25, 2023).

[35] Prometheus, “Alertmanager | Prometheus.” https://prometheus.io/docs/alerting/latest/alertmanager/ (accessed Mar. 02, 2023).
[36] Prometheus, “Exporters and integrations | Prometheus.” https://prometheus.io/docs/instrumenting/exporters/ (accessed Mar. 13, 2023).
[37] Grafana, “Plugin management | Grafana documentation,” *Grafana Labs*. https://grafana.com/docs/grafana/latest/administration/plugin-management/ (accessed Feb. 28, 2023).

Appendix 1 – Non-exclusive license for reproduction and publication of a graduation thesis[1]

I Aleksandr Borovkov

1. Grant Tallinn University of Technology free license (non-exclusive license) for my thesis “Development of a standardized set of checks for apps in OpenShift”, supervised by Aleksei Talisainen.
   1.1. to be reproduced for the purposes of preservation and electronic publication of the graduation thesis, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright;
   1.2. to be published via the web of Tallinn University of Technology, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright.
2. I am aware that the author also retains the rights specified in clause 1 of the non-exclusive license.
3. I confirm that granting the non-exclusive license does not infringe other persons' intellectual property rights, the rights arising from the Personal Data Protection Act or rights arising from other legislation.

24.04.2023

1 The non-exclusive licence is not valid during the validity of access restriction indicated in the student's application for restriction on access to the graduation thesis that has been signed by the school's dean, except in case of the university's right to reproduce the thesis for preservation purposes only. If a graduation thesis is based on the joint creative activity of two or more persons and the co-author(s) has/have not granted, by the set deadline, the student defending his/her graduation thesis consent to reproduce and publish the graduation thesis in compliance with clauses 1.1 and 1.2 of the non-exclusive licence, the non-exclusive license shall not be valid for the period.

Appendix 2 - Grafana kustomization file

```
---
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization

configurations:
  - kustomizeconfig/trigger.yaml

commonLabels:
  app: grafana

configMapGenerator:
  - name: grafana-env
    envs:
      - configs/grafana.env

secretGenerator:
  - name: grafana-build-webhook-secret
    literals:
      - "WebHookSecretKey=The webhook for grafana BuildConfig"
    type: Opaque

  - name: grafana-init-build-webhook-secret
    literals:
      - "WebHookSecretKey=The webhook for grafana_init BuildConfig"
    type: Opaque

resources:
  - git::https://******/kustomize_common_base.git/base?ref=master
  - resources/grafana.buildconfig.yaml
  - resources/grafana.deploymentconfig.yaml
  - resources/grafana.imagestream.yaml
  - resources/grafana.route.yaml
  - resources/grafana.service.yaml
  - resources/grafana-init.buildconfig.yaml
  - resources/grafana-init.imagestream.yaml
```

Appendix 3 – Grafana environment file

```
GF_DEFAULT_APP_MODE=production
GF_DEFAULT_INSTANCE_NAME=${HOSTNAME}
GF_PATHS_PROVISIONING=/etc/grafana/provisioning
GF_PATHS_PLUGINS=/var/lib/grafana_plugins

GF_SECURITY_DISABLE_GRAVATAR=true
GF_DATAPROXY_SEND_USER_HEADER=true

GF_SERVER_PROTOCOL=http
GF_SERVER_HTTP_PORT=3000
GF_SERVER_DOMAIN=localhost
GF_SERVER_ROOT_URL=%(protocol)s://%(domain)s:/
GF_SERVER_ROUTER_LOGGING=true
GF_SERVER_ENABLE_GZIP=true

GF_USERS_ALLOW_SIGN_UP=false
GF_USERS_ALLOW_ORG_CREATE=false
GF_USERS_AUTO_ASSIGN_ORG=true
GF_USERS_LOGIN_HINT=username
GF_USERS_DEFAULT_THEME=dark

GF_AUTH_ANONYMOUS_ENABLED=true
GF_AUTH_ANONYMOUS_ORG_NAME=Main Org.
GF_AUTH_ANONYMOUS_ORG_ROLE=Viewer

GF_AUTH_GITHUB_ENABLED=false
GF_AUTH_GOOGLE_ENABLED=false
GF_AUTH_GENERIC_OAUTH_ENABLED=false
GF_AUTH_GRAFANA_COM_ENABLED=false
GF_AUTH_PROXY_ENABLED=false
GF_AUTH_BASIC_ENABLED=false

GF_AUTH_LDAP_ENABLED=true
GF_AUTH_LDAP_CONFIG_FILE=/etc/grafana/provisioning/ldap.toml

GF_LOG_LEVEL=debug
GF_PANELS_DISABLE_SANITIZE_HTML=true
GF_ALERTING_ENABLED=false

GF_ANALYTICS_REPORTING_ENABLED=false
GF_METRICS_ENABLED=false
```

Appendix 4 – Grafana init Dockerfile

```
FROM ******/alpine:3.15.0

LABEL org.opencontainers.image.authors=""
LABEL org.opencontainers.image.vendor=""
LABEL org.opencontainers.image.title="Grafana configs image"
LABEL org.opencontainers.image.description="Grafana init-container image for populating grafana data volume"
LABEL org.opencontainers.image.base.name="******/alpine:3.15.0"

COPY ./src /src

VOLUME /data
VOLUME /provisioning

CMD cp -arv /src/provisioning/. /provisioning/ && \
    cp -arv /src/dashboards /data/
```

Appendix 5 – OpenShift monitor kustomization file

```
---
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization

commonLabels:
  service: Monitoring-Service
  project: Openshift-Monitor

bases:
  - git::https://******/kustomize_openshift_monitor_base.git/base?ref=master

resources:
  - grafana/
```

Appendix 6 – Alertmanager final configuration file

```
global:
  smtp_smarthost: 'smtp-******:25'
  smtp_from: 'DevOps Alert <noreply@******.com>'
  smtp_require_tls: false
route:
  group_by: ['alertname', 'severity']
  group_wait: 10s
  repeat_interval: 20m
  receiver: 'devops-team'
  routes:
    - receiver: 'devops-team'
      repeat_interval: 15m
      match_re:
        alert_group: openshift
receivers:
  - name: 'devops-team'
```

Appendix 7 – Kustomize configuration file in Working project folder

```yaml
---
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization

commonLabels:
  service: Monitoring-Service
  project: Openshift-Monitor

bases:
  - git::https://******/kustomize_openshift_monitor_base.git/base?ref=master

resources:
  - grafana/
```

Appendix 8 – Grafana kustomize configuration file in Working project folder

```
---
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization
namespace: ******

commonLabels:
  service: grafana
  project: Monitor_Openshift

patchesJson6902:

  - patch: |-
      - op: replace
        path: /spec/source
        value:
          contextDir: base/grafana/dockerfiles/initcontainer
          git:
            ref: master
            uri: ssh://git@******/kustomize_monitor.git
          sourceSecret:
            name: ******
          type: Git
    target:
      kind: BuildConfig
      group: build.openshift.io
      version: v1
      name: grafana-init

configMapGenerator:
  - name: grafana-env
    behavior: merge
    envs:
      - configs/grafana.env
resources:
  - git::https://******/kustomize_grafana_base.git/base?ref=master
```