MASARYK UNIVERSITY

FACULTY OF INFORMATICS

Collection and Evaluation of Monitoring Data from the Kubernetes Environment

Bachelor's Thesis

JAN BRÁZDA

Advisor: doc. Ing. Pavel Čeleda, Ph.D.

Department of Computer Systems and Communications

Brno, Spring 2023


FACULTAS ARTIS INFORMATICÆ
UNIVERSITAS MASARYKIANA

Declaration

Hereby I declare that this thesis is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Jan Brázda

Advisor: doc. Ing. Pavel Čeleda, Ph.D.

Acknowledgements

I would like to thank doc. Ing. Pavel Čeleda, Ph.D. for supervising my thesis and providing me with valuable advice.

Abstract

This thesis is focused on monitoring Kubernetes environments. The first three chapters provide an overview of the theory behind monitoring Kubernetes clusters, including a discussion of available monitoring tools, both specific and general-purpose. The fourth chapter of the thesis focuses on the implementation of a monitoring framework using the Elastic Stack, with a detailed description of the design and implementation steps. Finally, the effectiveness of the framework is evaluated to demonstrate its functionality and potential for improvement.

Keywords

Kubernetes, Docker, monitoring, cloud computing, Elastic, Prometheus, network traffic

Contents

Introduction

As cloud technologies gain popularity, Kubernetes has become the industry standard tool for container orchestration. Kubernetes can manage a large number of containers and scale according to the user's needs. However, that can lead to increased complexity, and it can become challenging to maintain the health and performance of applications running inside Kubernetes.

To address these issues, monitoring tools specifically adjusted for Kubernetes environments have been developed. These tools provide valuable insight into resource utilization, application behavior, or the overall health of the environment. To achieve that, monitoring tools need to read information from log files, collect metrics exported by applications, or fetch the data directly from Kubernetes API.

This thesis explains the process of collecting and evaluating monitoring data inside a Kubernetes cluster. It lists and compares some of the tools commonly used to monitor Kubernetes, such as Kubeshark [1], Prometheus [2], or Elastic Stack [3]. It also contains a proof of concept implementation of a monitoring framework for a Kubernetes cluster and evaluates its performance.

Chapter 1 describes the architecture of Kubernetes and the basics of monitoring it. Chapter 2 lists and compares tools that serve a specific purpose, like network traffic analysis or distributed tracing. Chapter 3 compares two main solutions to general monitoring, Prometheus stack, and Elastic Stack. Chapter 4 explains the requirements a monitoring framework needs to meet and describes its implementation. Chapter 5 evaluates if the implementation meets all the requirements.

1 Monitoring Kubernetes

Google [4] first introduced Kubernetes in 2014, and since then, it has grown to be among the most popular tools for software deployment [5]. It is currently developed and maintained by CNCF (Cloud Native Computing Foundation), a foundation aimed at advancing container technology [6].

1.1 Kubernetes

A microservice is an architectural style in which a number of independent processes communicate with each other. It gained popularity as an alternative to deploying monolithic applications [7]. Microservice architecture is often built using containers.

A container is a lightweight virtualization of an operating system, usually having resources to run only a single predefined process. Kubernetes was introduced as a way to deploy, manage, and scale containerized environments and facilitate communication between containers over the network, among other things.

Kubernetes controls an environment called a *cluster*, which consists of *Nodes*. To the users, a cluster appears as a single entity with the combined resources of all the involved machines. To control the cluster from the command line, *kubectl* can be used. It provides a command line interface to communicate with the Kubernetes API [8].

Nodes

A Node can be either a virtual machine or a physical computer. Nodes provide Kubernetes with the resources needed to run *Pods*.

Nodes have roles assigned to them. Every cluster needs at least one Node with a control plane role and one Node with a worker role.

Kubernetes components needed for managing the cluster are called a *control plane*, and they are run on Nodes with the control plane role. Worker Nodes are used to run the actual application workload [9]. Nodes can have both of these roles at the same time, allowing the existence of single-node clusters.

The control plane consists of these components:

- **Kube-apiserver** – exposes the Kubernetes API, allows managing the cluster with the kubectl command line tool,
- **Etcd** – stores key-value pairs containing cluster data,
- **Scheduler** – assigns Pods to available Nodes,
- **Controller-manager** – watches the state of the cluster and attempts to move it toward the desired state.

Additionally, every Node in the cluster has the following components:

- **Kubelet** – ensures that all desired containers are running and healthy,
- **Kube-proxy** – allows communication between Pods or Services inside a cluster,
- **Container runtime** – is a software that runs the containers, usually Docker.

An example of a Kubernetes cluster can be seen in Figure 1.1

Pods

In Kubernetes, a Pod is a wrapper over one or more containers [11]. A Pod can be seen as an abstraction of a physical host. Containers inside a Pod share the same file system and network namespace. Upon creation, the Pod is assigned a cluster-unique IP address. Containers within a Pod use the same port space and can communicate via localhost.

Pods are rarely created directly. Most of the time, they are deployed using a workload resource. One such resource, *Deployment*, is used to deploy a single Pod, while *ReplicaSet* spawns multiple instances of the specified Pod. *DaemonSet* runs the Pod on each Node in the cluster [12]. Workload resources, as any other resource in Kubernetes, can be specified using the YAML language.

Additional resources

Kubernetes uses many additional resources, but the most commonly seen are these:

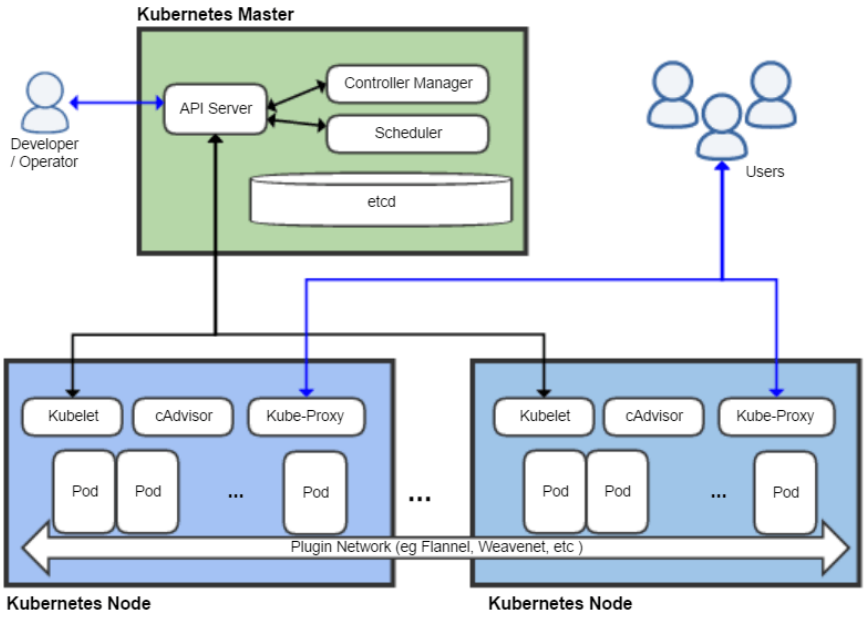

Figure 1.1: Overview of Kubernetes cluster [10]

- **Service** – assigns a static name and IP address to a Pod or groups of Pods,
- **Namespace** – groups related resources together,
- **Custom resource** – allows creating custom resources by extending the Kubernetes API,
- **Operator** – acts as a controller for a custom resource.

Authentication and Authorization

Kubernetes resources frequently disclose sensitive information, and it is necessary to limit access to those. To achieve this, Kubernetes provides two authorization mechanisms – RBAC (Role Based Access Control) and ABAC (Attribute Based Access Control).

RBAC grants access to the users based on the permissions of their roles. A *Role* is a Kubernetes object specifying rules that grants permissions to access resources. A Role grants permissions only within a

specific namespace; a *ClusterRole* grants them cluster-wide. *RoleBinding* and *ClusterRoleBinding* bind these roles to subjects, such as groups, users, or *ServiceAccounts*.

ABAC, on the other hand, uses policy objects. Each policy object matches a group of subjects and assigns them attributes. In Kubernetes, ABAC is defined using a JSON file [13].

1.2 Monitoring Kubernetes cluster

As clusters grow large, monitoring and managing them becomes complex [14], as it is harder to distinguish noise from relevant data. One of the most important metrics to observe is system resource usage, such as CPU usage or memory utilization. A *metrics server* [15] needs to be installed to obtain these metrics from Nodes and Pods inside the Kubernetes cluster. Once installed, the metrics can be inspected via Metrics API or using the kubectl command line tool.

Pods and containers encapsulated within them can be in a number of different states. From the perspective of monitoring them, Pods and containers are considered healthy if they are in a running state, and unhealthy otherwise. Pod downtime is calculated as a percentage of time the Pod has been unhealthy within a specified time frame.

Kubernetes creates a virtual network to enable communication between Pods, Services, and the Kubernetes API. Following this traffic flow on the packet level shows the IP addresses of hosts communicating with each other and the protocol over which they communicate. This is vital information for debugging or detecting suspicious activity. This aspect of monitoring clusters is described in Section 2.2.

The performance and health of crucial components of the cluster, such as kube-apiserves or kubelet should be closely monitored as well. Most components export these metrics by default, and they can be accessed on the `https://localhost:6443/metrics` endpoint. Accessing these metrics requires privileges that can be granted via RBAC or ABAC.

1.3 Collecting application data

Another aspect of monitoring Kubernetes clusters is monitoring the applications running inside them. Generally, the data related to the application's status is either written to a log file or exported as a metric.

Logs

Logs represent events that occurred while the application was running. The preferred way to store logs in Kubernetes is to write them to *stdout* or *stderr* [16], where logs will get collected by the container runtime.

Some applications might refuse to write logs to stdout or stderr [17] and only allow writing logs to regular files. A common solution to this problem is to deploy a sidecar container in the same Pod, grant it privileges to read the log files, and have this sidecar container write the logs to its own stdout, where it will get collected by a log collector [18].

Metrics

Metrics are numerical values generated by the application. There are two approaches to delivering metrics to the central system.

In a push-based system, applications are responsible for sending metrics to the central server. In a pull-based system, applications expose their metrics on an endpoint, and the agent periodically scrapes them.

Application performance

Application performance monitoring(APM) is a set of metrics that observe the application from the user's point of view. The number of requests, transactions, or average response time falls under this category.

2 Specific purpose monitoring tools

In this chapter, I will show tools for monitoring specific parts of Kubernetes environments. Each section is going to focus on one such part, and if multiple tools focus on similar tasks, I will compare these tools and showcase situations in which a particular tool may be used.

2.1 Cluster overview

Tools mentioned in this section focus on providing a general overview of the Kubernetes cluster as described in Section 1.2. In addition, they often provide options to manage the cluster, for example, adding, deleting, or editing Kubernetes objects, if the user operating them has sufficient privileges. However, they are not supposed to monitor application data. This will be handled by tools shown later in this chapter.

Kubernetes Dashboard

Dashboard is a web-based UI developed by the Kubernetes team. It is not deployed by default, but the installation is fairly straightforward. To access it, the user needs to have a service account with the *cluster-admin* role and generate a token unique to this account [19].

Dashboard lists all Kubernetes resources in a cluster, and information about them, like status, number of restarts, and date of creation. It also allows the creation of new resources using YAML, JSON, or by using a built-in form.

Dashboard fetches metrics regarding CPU and memory usage of workload resources and Nodes from the metrics server.

Lens Desktop Core

Lens Desktop Core, also known as OpenLens, is an open-source version of Lens Desktop, a standalone app developed by Mirantis, Inc. [20]. It has similar capabilities as Dashboard when it comes to monitoring and editing Kubernetes objects.

At the moment, only the source code of OpenLens is available in the GitHub repository, with no instructions on how to build it into an executable. This led to the creation of another repository containing instructions on how to build the source code and built binaries [21].

Comparison

While both of the above-mentioned tools are open-source, the deployment of OpenLens is significantly more complicated unless using binaries built by third-party, which introduces security risks.

These tools provide similar features when it comes to listing Kubernetes objects and their status. However, they differ in their approach to collecting system resource usage metrics. Dashboard uses metrics server, while OpenLens relies on a Prometheus instance installed in the cluster. This allows OpenLens to collect additional metrics, like filesystem reads and writes or network traffic, but introduces significant overhead unless a Prometheus instance is already installed.

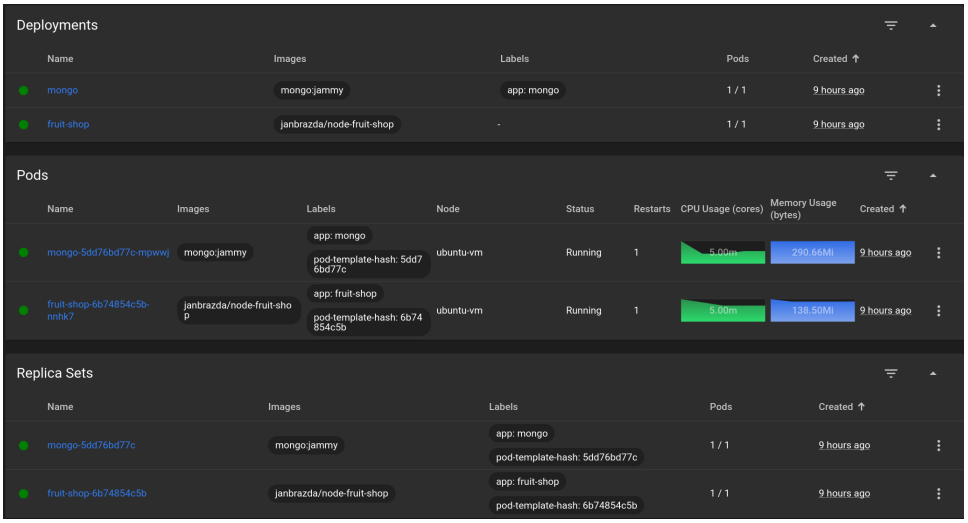

Figure 2.1: Overview of workloads in the target-3 namespace

OpenLens has useful additional features. From the UI, it allows the installation and managing Helm releases. Helm is a package manager for Kubernetes, allowing the deployment of complex Kubernetes

resources using just a few commands [22]. It also allows managing multiple clusters within one instance. Dashboard needs a separate instance for each cluster.

Considering the above-mentioned, Dashboard will be sufficient to provide an overview of a cluster in most situations. OpenLens should be considered if there is a need to monitor multiple clusters or a plan to use Helm extensively. An overview of workloads in a single namespace using Dashboard can be seen in Figure 2.1.

2.2 Network traffic

Tools mentioned in this section perform network traffic analysis. Network traffic analysis is usually done by inspecting individual packets and extracting information from them, like source and destination IP address, port, or protocol.

Ksniff

Knsiff is an open-source plugin for kubectl that allows capturing network traffic on a specific Pod inside the Kubernetes cluster [23]. The recommended way to install Ksniff is by using krew, a package manager for kubectl plugins.

By default, Ksniff will utilize tcpdump [24] to capture the traffic and Wireshark[25] to visualize it. There is an option to output the captured traffic to a file or stdout and process it from there. The user can also specify which container to capture the traffic on. By default, the first container in a Pod is chosen.

Ksniff will use kubectl to upload a compiled binary of tcpdump into a container and then redirect the traffic to the specified output. If the user running inside the container does not have enough privileges to capture the network traffic, Ksniff can be run in a privileged mode. In this mode, Ksniff will create a temporary Pod with access to the Docker daemon in the same namespace as the target Pod and perform the packet capture. Once the capture is complete, the Pod will get deleted.

It should be noted that according to the developers of Ksniff, the tool is not yet ready for production use [23].

Kubeshark

Kubeshark [1] is an open-source tool for monitoring and visualizing API traffic inside the cluster. It can capture part of network traffic in the Kubernetes cluster, namely the following protocols:

- HTTP/HTTPS,
- DNS,
- Redis,
- AMQP,
- Apache Kafka.

When deployed, Kubeshark creates its own namespace, and inside it a Pod that runs a frontend UI, a worker DaemonSet that captures the traffic on every Node, and finally, a hub that collects the data from workers and sends it over to the frontend.

Kubeshark allows capturing traffic on a single Pod, a group of Pods whose names match a regular expression, a whole namespace, or every Pod in a cluster.

The *service map* is a useful feature in Kubeshark for visualizing the overall traffic inside a selected part of the cluster. It is a directed graph where each service is a node, and edges indicate the bandwidth between the two nodes. An example of a service map can be seen in Figure 2.2.

Comparison

To capture and query network traffic, Ksniff uses industry-standard tools like tcpdump, Wireshark, or tshark, while Kubeshark comes with its own UI and backend, which means it can be used out of the box. However, someone who is already familiar with Wireshark might appreciate Ksniff more.

The limiting factor of Knsiff is that it can only capture traffic on one Pod at a time. Kubeshark, on the other hand, captures traffic on an arbitrary number of Pods. Ksniff, however, provides a more in-depth look into the traffic.

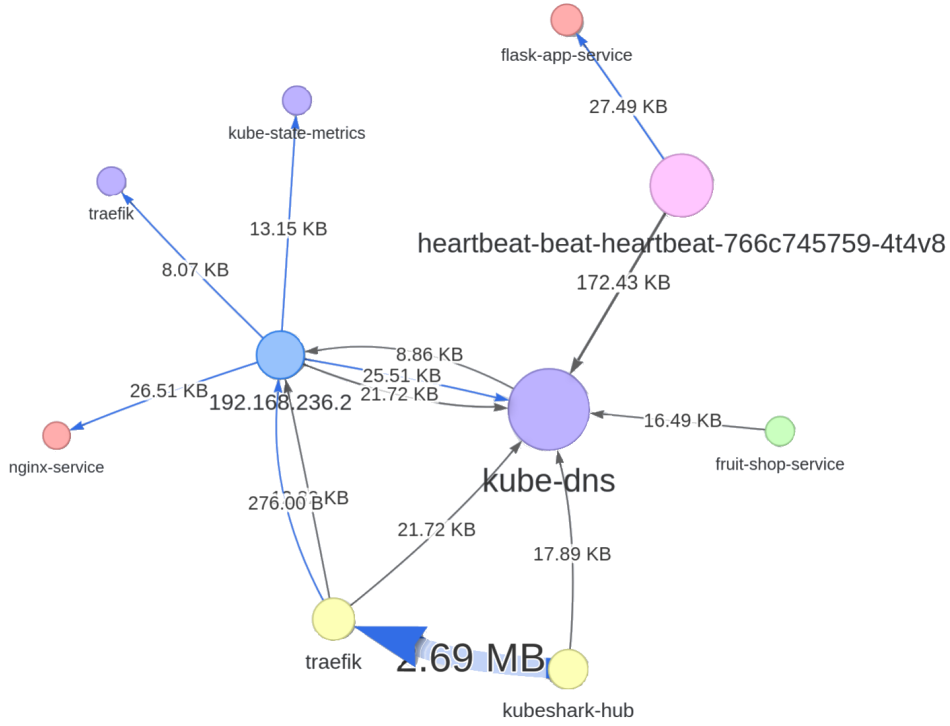

Figure 2.2: Service map showing traffic in a cluster

Kubeshark currently captures traffic from only a few protocols. If we are interested in traffic outside of these protocols, we might use Ksniff and get a deeper insight. Kubeshark however plans to support more protocols in the future, according to its official website.

2.3 Application performance

In this section, I want to show tools that monitor application performance that can be run inside a Kubernetes cluster.

Jaeger

Jaeger is an open-source distributed tracing system [26]. In a microservice architecture, it is common to have a vast amount of services communicating with each other. A single request from the user can travel

through multiple services, and if one service encounters a problem, it may be complicated to track it down. Jaeger provides insight into where the problem may have occurred.

Trace is a reconstruction of the path the request traveled through the system. Trace consists of spans. *Span* is a logical unit of work defined by the user. Each span has a name, duration, and, optionally, a parent span or child span. Trace can be seen as a directed acyclic graph, where nodes are represented by spans [27].

The inter-service communication is achieved with HTTP headers, where headers contain the current trace id, parent span id, and span id. This allows for the reconstruction of the trace once the request is completed.

To deploy the Jaeger backend, we will be using the Jeager operator for Kubernetes, which is going to manage the Jaeger instances. There are multiple strategies to deploy Jaeger. For testing purposes, it is recommended to use the all-in-one strategy.

```
app = Flask(__name__)
@app.route("/user_info", methods=["GET"])
@tracer.start_as_current_span("GET␣/
user_info")
def info():
    with tracer.start_as_current_span("mysql
␣connection"):
        cur = mysql.connection.cursor()
    with tracer.start_as_current_span("fetch
␣users"):
        cur.execute("""SELECT * FROM users
""")
        rv = cur.fetchall()
    return str(rv)
```

Figure 2.3: Python code implementing Jaeger tracing

The code snippet in Figure 2.3 shows an implementation of tracing a Python Flask application. The trace will start when a request is made

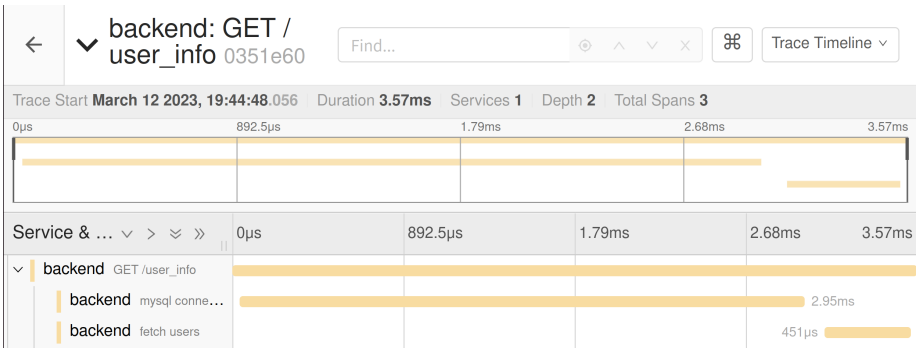

Figure 2.4: Trace detail in the Jaeger UI

to `user_info` endpoint and then has two child spans, one to connect to the database and a second to execute a SQL query. The resulting trace, as shown in the Jaeger UI, can be seen in Figure 2.4.

3 General purpose monitoring tools

This chapter describes tools used to monitor most of the aspects of the Kubernetes environment, combining data from the Kubernetes cluster and applications running inside it. To achieve that, these tools provide a high level of customization. The most suitable solutions for the general monitoring of a cluster will be described below, and part of this chapter is dedicated to comparing them.

3.1 Prometheus stack

Prometheus stack has two main building blocks, Prometheus and Grafana. It can be extended with Loki [28] and Promtail [29] to monitor log files.

Prometheus

Prometheus is an open-source tool that collects metrics from various systems and services. It is a pull-based system and stores the data in a time series format. To search the stored data, it uses a PromQL query language. Alertmanager can be used to send alerts based on alerting rules specified in Prometheus.

Grafana

Grafana is a visualization technology that provides interactive dashboards. It has native support for Prometheus, and it is, therefore, easy to integrate these tools together.

How it works

Prometheus periodically scrapes the `http://<domain_name>/metrics` endpoint on targets that it discovers. In the Kubernetes environment, these targets will usually be Services, Pods, or Nodes. To make Prometheus aware of these targets, we can either specify the static URL of a target or use Kubernetes auto-discovery, which will fetch the targets from the Kubernetes API server. This can be combined

with a matching label mechanism, where Prometheus will scrape only resources with a specific label, for example, `"prometheus.io/scrape"=true`.

As mentioned, applications need to expose the metrics over HTTP or HTTPS. Some applications will do this by default, but others need to be set up manually. This is usually done with client libraries. Prometheus offers official support for the most commonly used programming languages used today, and community-maintained libraries are available for others. Additionally, Prometheus offers exporters for third-party applications like databases or web servers [30]. The exporters will automatically export metrics relevant to the application and might even provide pre-built Grafana dashboards.

Once Prometheus scrapes the metrics, it saves them to local or remote storage, depending on the user's preferences. These metrics can then be explored with Prometheus web UI, Grafana, or using an API.

Logs

Since Prometheus lacks native support for collecting logs, we need to install additional tools to achieve this.

Promtail is an agent responsible for collecting the contents of application logs and, therefore, should be run as a DaemonSet to ensure it collects logs from all Nodes.

Loki is a log aggregation system that ingests the logs sent by various agents, like Promtail, in this case. Both of these tools are developed by Grafana Labs and are, therefore, easily integrated with Grafana.

3.2 Elastic Stack

Elastic Stack (also known as ELK Stack) consists of tools developed by the Elastic company. Traditionally the components of the stack are:

- **Elasticsearch** – stores the data and allows searching it,
- **Kibana** – visualization tool providing interactive dashboards using data from Elasticsearch,

- **Logstash** – receives the data from input (Beats), transforms it, and sends it to output (Elasticsearch),

- **Beats** – collects and sends data to Logstash (or Elasticsearch directly), including Filebeat for reading contents of log files, Metricbeat for collecting metrics, Packetbeat for collection of network information, and others.

A visualization of components of the Elastic Stack can be seen in Figure 3.1.

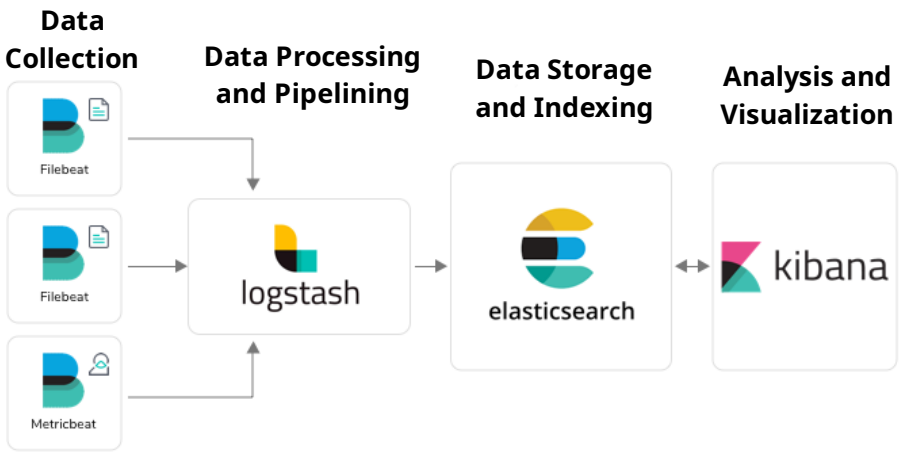

Figure 3.1: Architecture of Elastic Stack [31]

Elastic Cloud on Kubernetes

ECK (Elastic Cloud on Kubernetes) is an operator meant to simplify the deployment and management of Elastic Stack on Kubernetes. It makes the orchestration of the stack almost effortless. While using ECK, components can be easily connected to each other, and their communication is secured by TLS certificates. Additionally, ECK makes it much easier to upgrade components of the stack to a newer version [32].

Elastic agents and Fleet server

The Elastic agent is a recently introduced alternative to Beats. It provides the same functionality as the various Beats do, but as a single entity. In the Kubernetes environment, each Beat would have to be run as a separate Pod, while the agent needs just one Pod.

Agents will collect data depending on what integrations are specified in their policy. Integrations define how to observe a data source and provide a way to parse the raw data to a format that Elasticsearch can understand. They often come packaged with Kibana visualizations. There are integrations for most of the popular applications, such as databases, web servers, or cloud management tools.

Each agent needs to be signed to an agent policy. Agent policy is a list of integrations and their relative configurations.

Agents can run in standalone mode or be managed by Fleet. In standalone mode, the user has to install the agent on the host and is responsible for upgrading and managing it. When managed by Fleet, agents have to be connected to the Fleet server, which they will periodically check for changes to their policy and update their configuration accordingly. Fleet can be easily managed from Kibana UI. Elastic recommends running agents in Fleet, as the standalone mode is reserved for advanced users and not well documented [33].

3.3 Comparison

This section compares the Elastic Stack to the Prometheus stack.

Licensing

Prometheus, Grafana, and other components of the stack have been open-source since their release, and Prometheus is a graduated CNCF project, same as Kubernetes [34].

Elastic Stack, on the other hand, used to be open-source until January 2021, when Elastic announced a change of the license from Apache 2.0 to a dual license of SSPL and Elastic license [35]. In this case, dual licensing means that users can use the license that fits their use case more. The Elastic license restricts users from providing the products under this license as managed services and bypassing its

features protected by the license key. SSPL license allows offering the provided software as managed service. However, anyone who does so must provide the source code of the service that uses such software.

This change was done mainly to stop cloud providers like Amazon AWS and Microsoft Azure from offering managed Elastic Stack service to their customers. It should not impact users wanting to use the Elastic Stack to monitor their infrastructure.

Elastic offers the management of the monitoring stack on major cloud providers for a monthly payment, or users can self-manage the stack for free. When self-managing the stack, the core features are free. However, some premium features, for example anomaly detection using machine learning, require a subscription.

Scope

Prometheus itself can not track logs, and we need to add Loki to achieve that. Elastic Stack, on the other hand, focuses mainly on logs. It does have support for metrics using Metricbeat, and users can even export their own metrics from applications using APM agents [36].

Indexing

Elasticsearch stores its data as JSON documents, and it uses a data structure called an inverted index – it stores each unique word that appears in a document and also identifies all documents that word appears in – this allows for very fast full-text searching [37], but might lead to degradation of performance at large scale [38].

Prometheus and Loki chose a different approach, where they index only the metadata of the document but not the actual contents. This approach requires less storage and might be faster to run, but it lacks the ability to query the contents of logs, which is a significant downside.

Hardware requirements

Due to the way these tools index the data, Elastic Stack requires more storage to store its data.

Table 3.1: Memory limits of Elastic Stack components [39]

Component	Memory limit
APM server	512 MiB
Elasticsearch	2 GiB
Kibana	1 GiB
Elastic Agent	350 MiB

Table 3.2: Memory limits of Prometheus stack components [40, 41]

Component	Memory limit
Prometheus	400 MiB
Loki	128 MiB
Grafana	128 MiB
Promtail	128 MiB

The default memory limits, set by the Elastic operator, are shown in Table 3.1. I could not find official recommendations for the Prometheus stack, but I managed to collect the values from the Helm charts of example deployments. These limits are shown in Table 3.2. Evidently, the Elastic Stack has higher hardware requirements.

Features

Elastic Stack has additional features that Prometheus stack lacks. For example, the APM offers distributed tracing, as described in Section 2.3. Prometheus stack would need to be connected to a Jaeger instance to perform distributed tracing.

Elastic offers *Network Packet Capture* integration that provides detailed information about network traffic and flows, and it comes with useful visualizations, while Prometheus stack has no built-in features for analyzing network traffic.

The *Defend* integration has many features relevant to security, such as malware detection, threat detection, and the history of commands and processes that have run on a system. There is currently an issue running this integration on the same host as the Elastic agent. However, it should be resolved in the upcoming version 8.8 [42].

4 Design and implementation of monitoring framework

This chapter summarizes the requirements that a monitoring framework for Kubernetes needs to meet. It then describes a proof of concept solution. The files I will be referencing can be found in Attachment A.

4.1 Functional requirements

The monitoring framework is required to monitor certain aspects of the Kubernetes environment. This section is going to describe these aspects in detail.

Kubernetes objects

The monitoring framework should be able to interact with Kubernetes APIs to collect data about the state of the cluster, its Nodes, and its workloads. It should also be able to monitor the resource utilization of Kubernetes objects.

Container runtime

The framework should monitor the state of the containers running inside the Pods in the Kubernetes environment. The information can be gathered from logs or metrics generated by the container runtime.

Log files

The framework should collect and parse the contents of log files. It should collect system logs, which might include information about login attempts or users running commands with elevated privileges. In addition to that, it is necessary for any framework to also easily parse the log files of commonly used applications, like web servers or databases.

Application performance monitoring

APM might include metrics like response time, throughput, or error rates. The framework should also have some capabilities when it comes to distributed tracing.

Network traffic

The monitoring framework should discover the network's topology, including all the communicating hosts. It should be able to capture and analyze the traffic and decode some of the commonly used protocols.

4.2 Kubernetes cluster

Before implementing the proof of concept, it is necessary to create a Kubernetes cluster. For this purpose, I chose k3s, a lightweight Kubernetes distribution [43]. Creating and starting the cluster is as simple as running a single bash command:

```
curl -sfL https://get.k3s.io | K3S_KUBECONFIG_MODE="644"
sh -s - --docker
```

The `K3S_KUBECONFIG_MODE="644"` option sets the permissions of the kubeconfig file, allowing non-root users to interact with the cluster, and the `--docker` option tells k3s to run containers using the Docker engine, instead of containerd, which is the default one.

At this point, it is probably a good time to install *kube-state-metrics*. Kube-state-metrics is a service that listens to the Kubernetes API and exports metrics about individual objects [44]. The Elastic Stack requires this to provide observability to the Kubernetes cluster. A sample configuration is located in the `monitoring/kube-state-metrics` directory in Attachment A.

4.3 Sample applications

In order to generate some traffic inside the cluster, I decided to create three sample applications. These applications consist of commonly used software and demonstrate some techniques used in monitoring application data. Each application has its own namespace.

Table 4.1: URLs of sample applications

Application	URL
Target-1	http://localhost:32001
Target-2	http://localhost:32002
Target-3	http://localhost:32003

The source code of these applications can be found in the `targets/` directory, and they can be installed by running the `scripts/install_targets.sh` script. The applications are exposed via a NodePort Service and are accessible at ports shown in Table 4.1.

There are two custom dashboards in the `dashboards` directory. One is for visualizing the data from *target-3*, and the second one is for visualizing the number of bytes transferred between IP addresses in the cluster. The dashboards can be imported to Kibana using the `scripts/dashboards.sh` script.

Target-1

The *target-1* namespace contains two Pods, a MySQL database and a WordPress web application. MySQL is configured to write error and slow-query logs to the `/var/log/` directory, which gets mounted to the host to ease the collection of these logs.

Target-1 shows how to configure the MySQL integration of Elastic Stack.

Target-2

There are two Pods in the *target-2* namespace, a Flask web application written in Python and a nginx proxy.

The nginx proxy listens on port 80. It uses the nginx *stub_status* module to expose basic statistics on the `nginx_status` endpoint and redirects requests to all other endpoints to the Flask application. Nginx logs are written to the `/var/log` directory.

The Flask application simulates a distributed transaction on the `http://flask-app-service:5000/transcation` endpoint. When a user makes a request to the */transcation* endpoint, it waits for a random amount of time and then makes a request to the `http://fl`

ask-app-service:5000/part-one endpoint. This endpoint will then call another endpoint http://flask-app-service:5000/part-two a random number of times. The */part-two* endpoint will then return either HTTP 200 response code or again wait for a small amount of time and return an HTTP 500 error code. This process is supposed to imitate an internal problem within an application.

Additionally, the application exposes the http://flask-app-service:5000/status endpoint, which responds with an HTTP 500 error code by default, but the user can change this behavior by clicking a button, which will make the endpoint respond with an HTTP 200 success code.

The application uses a Python library to connect to the Elastic APM server and send all of the data mentioned above.

Target-3

Target-3 namespace imitates a website that sells fruit and allows users to purchase and ask for the restocking of different kinds of fruit. The backend is written in JavaScript and connects to a MongoDB database, where it fetches information about current stock. The application writes information to logs with a custom format, where each log contains an IP address of a request, status, and message. An example of the contents of the log file can be seen in Figure 4.2.

APM Node.js agent library is used to export custom metrics to the APM server. The URL of of the APM server is passed in as an environmental variable.

To parse the logs created by this application, it uses Custom Logs integration. To achieve that, it needs a custom ingest pipeline. The script used to import the pipeline to Kibana is `scripts/fruit_shop_logs_pipeline.sh`.

This application is supposed to show how the Elastic Stack can parse custom log files using Custom Logs integration. It also exports custom metrics to the APM server and creates its own Kibana dashboard. Figure 4.1 shows a preview of the application.

Figure 4.1: Target-3 application

```
::ffff:10.42.0.1 SUCCESS Ordered 125 of
   strawberry
::ffff:10.42.0.1 SUCCESS Restocked
::ffff:10.42.0.1 SUCCESS Restocked apple
::ffff:10.42.0.1 FAILURE Tried to order 155 of
   banana when 64 are available
```

Figure 4.2: Contents of target-3 log file

4.4 Choosing the monitoring framework

The two monitoring solutions described in Chapter 3, Elastic Stack and Prometheus stack, are both capable of monitoring a Kubernetes environment. I decided to implement Elastic Stack as a proof of concept solution because it is more capable than the Prometheus stack in collecting data from log files, which is one of the most important sources of information in a Kubernetes cluster. Additionally, it provides dashboards for network traffic analysis, a feature that the Prometheus stack lacks.

4.5 Hardware and software requirements

There are certain requirements for a system to run this proof of concept solution successfully. It should be run on a Linux system with at least 10 GB of RAM and 20 GB of free storage. The system should have Docker and curl [45] installed. The system should also have access to the Internet.

I tested the deployment on Ubuntu 22.04 system, and the versions of the used software are shown in Table 4.2.

Table 4.2: Used software and its version

Software	Version
k3s	1.26.4
Kubernetes	1.25.4
Elastic Stack	8.6.1
Docker	20.10.21

4.6 Implementing Elastic Stack

One of the first steps in implementing the Elastic Stack is usually going to be installing ECK (Elastic Cloud on Kubernetes). While ECK is not necessary to deploy the Elastic Stack, it significantly eases its deployment and management. The YAML files necessary for installing ECK are located in the `monitoring/eck` directory.

Now, with ECK installed, it is easier to install the individual components of the Elastic Stack. There is a `scripts/install_elastic_stack.sh` script to install all the components at once.

Elasticsearch

The YAML configuration file for Elasticsearch is located at `monitoring/elastic_stack/elasticsearch.yaml`. The relevant setting here is the number of replicas of Elasticsearch, currently set to 1.

Kibana

The configuration file for Kibana is located at `monitoring/elastic_stack/kibana.yaml`. Thanks to ECK, connecting Kibana to the Elasticsearch instance is easily done by specifying the name of Elasticsearch deployment in the `spec.elasticsearchRef.name` attribute.

Additionally, most of the configuration for the Elastic agent can be specified in the Kibana config file, specifically in the `spec.config` section. These settings can also be configured using the Kibana UI. First, it is necessary to configure the Elasticsearch and Fleet server URLs the Agents should connect to, using the `xpack.fleet.agents.elasticsearch.hosts` and `xpack.fleet.agents.fleet_server.hosts` attributes, respectively. Integrations that should get installed are specified in the `xpack.fleet.packages` attribute. The agent policies are specified in the `xpack.fleet.agentPolicies` section. There are two policies, *Fleet Server* and *Elastic Agent*.

Each policy has a `package_policies` section, which is an array of integration names and their configurations. It is necessary to include the Fleet server integration in the Fleet server policy.

There are two additional Services to provide additional functionality to Kibana. The `monitoring/elastis_stack/kibana-svc.yaml` is *NodePort* Service. NodePort Services expose a Service on a static port on each Node inside the cluster, and this will simplify accessing Kibana. In this case, Kibana is accessible on `https://localhost:32000`.

The second Service is `monitoring/elastic_stack/kube-state-metrics-svc.yaml`. This Service is of type *ExternalName*, and it will map the kube-state-metrics from the *kube-system* namespace to the same namespace as Kibana. This is because the Kubernetes integration

needs access to these metrics, and it would be complicated to access them in different namespace.

Elastic agent and Fleet server

Configuration of Elastic agent is located at `monitoring/elastic_stack/elastic-agent.yaml`. Since this proof of concept is running locally, and all the log files are stored on the hard drive, it is necessary to mount the `/var/log` and `/var/lib/docker/containers` directories inside the agent so that it can access them. It also mounts the Docker socket, usually located at `/var/run/docker.sock`, to give the Docker integration access to metrics.

To ensure that the Agent uses the same timezone I am currently located in, I decided to mount the timezone file `/usr/share/zoneinfo/Europe/Prague` to `/etc/localtime` in the Agent filesystem.

The elastic agent needs a service account with elevated privileges to perform some tasks. This service account is defined in `monitoring/elastic_stack/elastic-agent-role.yaml`, along with its *ClusterRole* and *ClusterRoleBinding*

The Fleet server uses the same custom resource definition as the Agent. What enables the agent to function as a Fleet server is the attribute `spec.fleetServerEnabled` set to **true**. The files relevant for the Fleet server are `monitoring/elastic_stack/fleet-server.yaml` and `monitoring/elastic_stack/fleet-server-role.yaml`

Heartbeat

Heartbeat probes specified targets for their availability and health. It can use HTTP, TCP, or ICMP protocols. In the context of the Kubernetes cluster, it offers autodiscovery of Pods, Services, and Nodes. When autodiscovery of Kubernetes objects is configured, it will automatically detect existing targets and start probing them. Heartbeat also allows specifying a static address of a target. The monitors are available in the Uptime tab in Kibana. The relevant configuration files are `monitoring/elastic_stack/heartbeat.yaml` nad `monitoring/elastic_stack/heartbeat-role.yaml`.

APM server

Elastic APM server receives data from applications, transforms it, and then sends it to Elasticsearch. The configuration is available at `monitoring/elastic_stack/apm-server.yaml`.

5 Evaluation

This chapter presents the results of evaluating the monitoring framework designed and implemented in Chapter 4. The goal of this evaluation is to assess the effectiveness and efficiency of the framework in collecting and analyzing monitoring data from Kubernetes environments.

5.1 Satisfaction of requirements

This section is going to compare the functionality of the Elastic Stack to the requirements described in Section 4.1.

Kubernetes objects

Most of the information Elastic Stack acquires about the Kubernetes objects is via the Kubernetes integration. This integration comes with 15 dashboards, including ones for Pods, Services, or Nodes. A part of dashboard related to the overview of Kubernetes cluster can be seen in Figure 5.1. To maximize the utility of this integration, kube-state-metrics should be installed in the cluster and accessible by the Elastic agent.

Another insight can be gained by looking at the Infrastructure tab of Elastic Observability. This page shows the resource utilization of Pods or Nodes.

Elastic Heartbeat monitors targets for their availability and downtime. It also offers autodiscovery of Kubernetes Pods, Services, and Nodes. With this feature enabled, these Kubernetes objects will be monitored in the Uptime tab of Elastic Observability.

Container runtime

The container runtime used for this proof of concept is Docker. Elastic offers a Docker integration, which needs to connect to the Docker socket, usually located at `/var/run/docker.sock`. After that, the provided dashboard will show the resource utilization of the containers or the images they use.

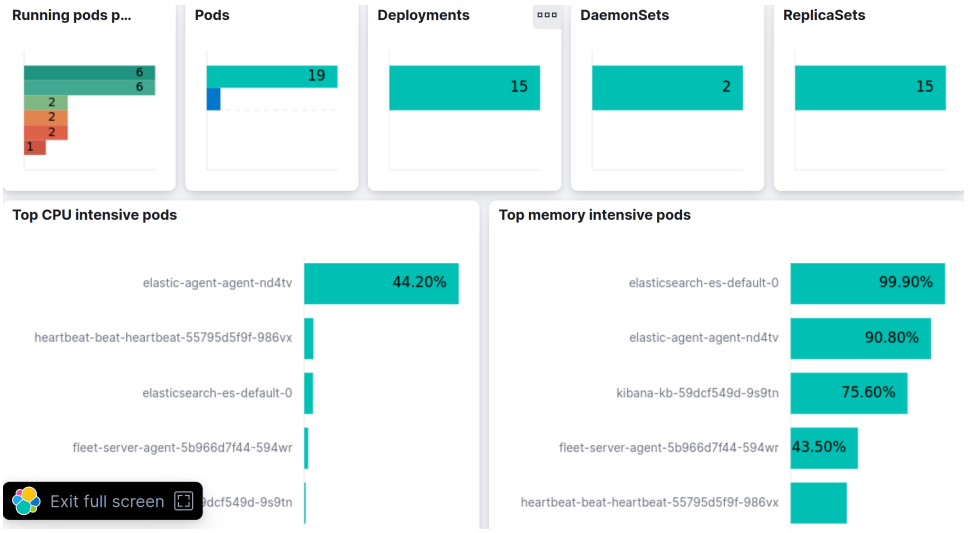

Figure 5.1: Part of a Kubernetes Dashboard in Kibana

Similarly to Kubernetes Pods, an overview of Docker containers can be seen on the Infrastructure tab.

Log files

Elastic offers hundreds of integrations for the most commonly used applications and their log files. These integrations contain dashboards and pipelines to parse the data. The proof of concept, which is a part of this thesis, shows the configuration for Nginx and MySQL integrations, in addition to Custom Logs, an integration to parse custom application logs.

For the system logs, Elastic uses System integration. This integration supports collecting data from both Windows and Linux systems. On Linux hosts, it collects information about user logins or the creation of new users from the authentication logs, usually located at `/var/log/auth.log`. An example of visualization for displaying SSH login attempts can be seen in Figure 5.2. It also reads the logs from processes running on the system from syslog. On Windows, it collects the Application, Security, and System logs [46].

@timestamp	system.auth.ssh.ev...	system.auth.ssh.me...	user.name	source.ip
May 3, 2023 @ 20:30:18.000	Accepted	password	jan	127.0.0.1
May 3, 2023 @ 20:29:27.000	Failed	password	test	127.0.0.1
May 3, 2023 @ 20:29:23.000	Invalid	-	test	127.0.0.1

Figure 5.2: SSH login attempts visualized by the System integration

The System integration also collects detailed metrics about resource utilization on each host.

APM

Elastic APM provides visibility into the application's performance. It provides some basic statistics, for example, the most visited endpoints and how often they return an error. An example of these statistics can be seen in Figure 5.3. It also offers a solution for distributed tracing, where it shows details about each trace.

For an application to send data to Elastic APM, it needs to connect to an APM server. This is usually done with client libraries that will also automatically send the needed data. There are libraries available for most of the popular programming languages.

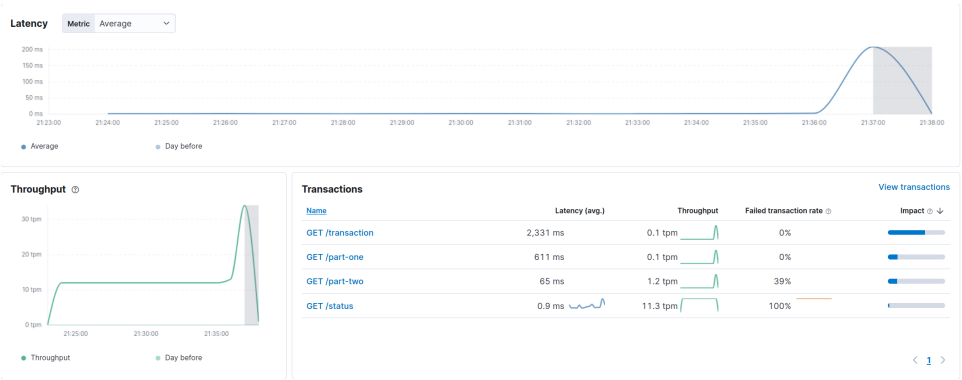

Figure 5.3: Applications performance shown in Elastic APM

Network traffic

To capture and analyze network traffic, Elastic uses Network Packet Capture integration. It recognizes the most popular network protocols, like DNS, TLS, HTTP, or ICMP, among others. The included dashboards show the overall analysis of the captured traffic, as well as a specific dashboard for most of the protocols it supports.

Elastic also offers a NetFlow records integration, which is able to receive NetFlow or IPFIX flow records and provide visualizations for them.

5.2 Scalability

As mentioned in Section 1.2, Kubernetes clusters may grow large over time and that means that the monitoring framework needs to handle more data. The Elastic Stack can run into issues if it is not properly scaled to the size of the cluster. This section will outline some of the challenges with scaling Elastic Stack.

Indexing

Elasticsearch indexes all the contents of a document, as mentioned in Section 1.2. Depending on the amount of data flowing through the cluster, storing and indexing documents in this way can put a lot of pressure on the resources of the Nodes Elasticsearch is running on. To deal with this, Elasticsearch splits the index into *shards*, where each shard contains a part of the original index. These shards are then distributed across Nodes, and when new Nodes get added to the cluster, Elasticsearch moves some shards into it to keep the balance [47]. By doing this, Elasticsearch lowers the hardware requirements of each Node.

Tuning JVM settings

Elasticsearch is built on top of Java, and it uses Java Virtual Machine (JVM) to run. Elastic recommends keeping the provided JVM options [48]. However, changing them can lead to increased performance.

One of the most relevant settings for performance is the heap size of the JVM. Increasing the size of the heap increases the memory Elasticsearch can use for its internal cache, but it leaves less memory for the operating system. Finding the optimal heap size can significantly increase performance.

Adding more instances

Currently, the framework implemented in Chapter 4 runs only one copy of Elasticsearch and Kibana. This is due to the fact that these components consume a lot of system resources. Increasing the number of instances will boost performance and increase reliability.

5.3 Potential improvements

Although the implemented framework satisfies all of the requirements, there are several potential improvements. Some of these improvements are discussed below.

Proxy to access Kibana

Kibana is exposed via a NodePort Service, which acts as a proxy into the Kubernetes cluster. This makes it easy to access it on the `https://localhost:32000` address. However, when accessing it, the browser will show a warning about a self-signed certificate. The visitor can choose to accept the risk and visit the site anyway. The user experience could be improved by installing a proxy with valid certificates, for example, nginx, in front of Kibana.

HostPath volumes

Currently, the log files are mounted into the Elastic agent Pod, where the agent reads them. This is achieved with a *hostPath* volumes. HostPath volumes mount directories and files from the host Node inside the container. HostPath volumes are a security risk, and even the Kubernetes documentation warns about using them [49].

A more secure solution would be to send the logs to remote storage and then let the Elastic agent access this storage.

Split the cluster into two

Currently, applications and the monitoring framework run in the same cluster. If the cluster itself starts malfunctioning, the monitoring tools will be unavailable and won't help with fixing the problem.

Because of that, it is common to have multiple clusters, one running the applications and the second one running the monitoring framework.

Conclusion

This thesis has examined the challenges of monitoring a Kubernetes environment and evaluated various monitoring tools and frameworks. Through this research, several key factors have been identified to consider when selecting a monitoring solution.

The thesis began by discussing the architecture of Kubernetes and the process of monitoring it. Tools for monitoring specific are of the environment, like network traffic or application performance, and general-purpose monitoring tools like Prometheus and Elastic Stack were then reviewed and compared.

Based on the comparison, Elastic Stack was selected as the monitoring framework for the proof of concept implementation. Elastic Stack offers flexibility, scalability, and powerful search and analysis capabilities necessary for effective monitoring.

The implementation was then evaluated to see if it met the outlined requirements. Some of the flaws of the implementation were identified and potential improvements were suggested.

Overall, this thesis has demonstrated the importance of monitoring a Kubernetes environment and the complexity of selecting an effective monitoring solution. Careful consideration of the factors outlined in this thesis can help in selecting a monitoring framework that meets specific needs and improves the overall health and performance of a Kubernetes environment.

Future research in this area could include further evaluation of other monitoring tools and frameworks, exploring the use of machine learning and other advanced analytics techniques for monitoring.

Bibliography

1. *kubeshark/kubeshark: The API traffic analyzer for Kubernetes providing real-time K8s protocol-level visibility, capturing and monitoring all traffic and payloads going in, out and across containers, pods, nodes and clusters.*. [online]. [visited on 2023-03-14]. Available from: https://github.com/kubeshark/kubeshark.
2. *What is Prometheus?* [online]. [visited on 2023-03-16]. Available from: https://prometheus.io/docs/introduction/overview/.
3. *Elastic Stack: Elasticsearch, Kibana, Beats & Logstash | Elastic* [online]. [visited on 2023-05-16]. Available from: https://www.elastic.co/elastic-stack/.
4. BURNS, Brendan et al. Borg, Omega, and Kubernetes. *Communications of the ACM*. 2016, vol. 59, no. 5, pp. 50–57. ISSN 0001-0782. Available from DOI: 10.1145/2890784.
5. *The State of Enterprise Open Source: A Red Hat report* [online]. [visited on 2023-03-07]. Available from: https://www.redhat.com/en/resources/state-of-enterprise-open-source-report-2022.
6. *New Cloud Native Computing Foundation to drive alignment among container technologies* [online]. [visited on 2023-03-12]. Available from: https://www.cncf.io/announcements/2015/06/21/new-cloud-native-computing-foundation-to-drive-alignment-among-container-technologies/.
7. DRAGONI, Nicola et al. Microservices: Yesterday, Today, and Tomorrow. In: *Present and Ulterior Software Engineering*. Ed. by MAZZARA, Manuel; MEYER, Bertrand. Cham: Springer International Publishing, 2017, pp. 195–216. ISBN 978-3-319-67425-4. Available from DOI: 10.1007/978-3-319-67425-4_12.
8. *Command line tool (kubectl)* [online]. [visited on 2023-03-12]. Available from: https://kubernetes.io/docs/reference/kubectl/.

9. *Kubernetes Components* [online]. [visited on 2023-04-08]. Available from: https://kubernetes.io/docs/concepts/overview/components/.
10. *Kubernetes* [online]. [visited on 2023-02-27]. Available from: https://upload.wikimedia.org/wikipedia/commons/b/be/Kubernetes.png.
11. *What are containers?* [online]. [visited on 2023-02-27]. Available from: https://www.ibm.com/topics/containers.
12. *Workloads* [online]. [visited on 2023-02-27]. Available from: https://kubernetes.io/docs/concepts/workloads/.
13. *ABAC* [online]. [visited on 2023-02-27]. Available from: https://kubernetes.io/docs/reference/access-authn-authz/abac/#policy-file-format.
14. SUKHIJA, Nitin; BAUTISTA, Elizabeth. Towards a Framework for Monitoring and Analyzing High Performance Computing Environments Using Kubernetes and Prometheus. In: *2019 IEEE SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI*. 2019, pp. 257–262. Available from DOI: 10.1109/SmartWorld-UIC-ATC-SCALCOM-IOP-SCI.2019.00087.
15. *Kubernetes Metrics Server* [online]. [visited on 2023-02-28]. Available from: https://github.com/kubernetes-sigs/metrics-server.
16. *How nodes handle container logs* [online]. [visited on 2023-03-08]. Available from: https://kubernetes.io/docs/concepts/cluster-administration/logging/#how-nodes-handle-container-logs.
17. *MySQL cannot write slow logs to /dev/stderr* [online]. [visited on 2023-03-08]. Available from: https://alexanderallen.medium.com/mysql-cannot-write-slow-logs-to-dev-stderr-3e8e6ac5c4f8.
18. *Using a sidecar container with the logging agent* [online]. [visited on 2023-03-08]. Available from: https://kubernetes.io/docs/concepts/cluster-administration/logging/#sidecar-container-with-logging-agent.

19. *Dashboard* [online]. [visited on 2023-03-06]. Available from: https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard/.

20. *lensapp/lens: Lens - The way the world runs Kubernetes* [online]. [visited on 2023-03-12]. Available from: https://github.com/lensapp/lens.

21. *MuhammedKalkan/OpenLens: OpenLens Binary Build Repository* [online]. [visited on 2023-03-06]. Available from: https://github.com/MuhammedKalkan/OpenLens.

22. *helm/helm: The Kubernetes Package Manager* [online]. [visited on 2023-04-13]. Available from: https://github.com/helm/helm.

23. *eldadru/ksniff: Kubectl plugin to ease sniffing on kubernetes pods using tcpdump and wireshark* [online]. [visited on 2023-03-13]. Available from: https://github.com/eldadru/ksniff.

24. *TCPDUMP & LIBPCAP* [online]. [visited on 2023-05-10]. Available from: https://www.tcpdump.org/.

25. *Wireshark · Go Deep* [online]. [visited on 2023-05-10]. Available from: https://www.wireshark.org/.

26. *jaegertracing/jaeger: CNCF Jaeger, a Distributed Tracing Platform* [online]. [visited on 2023-03-12]. Available from: https://github.com/jaegertracing/jaeger.

27. *Trace* [online]. [visited on 2023-03-13]. Available from: https://www.jaegertracing.io/docs/1.42/architecture/#trace.

28. *grafana/loki: Like Prometheus, but for logs.* [online]. [visited on 2023-05-11]. Available from: https://github.com/grafana/loki.

29. *Promtail* [online]. [visited on 2023-05-11]. Available from: https://grafana.com/docs/loki/latest/clients/promtail/.

30. *EXPORTERS AND INTEGRATIONS* [online]. [visited on 2023-03-17]. Available from: https://prometheus.io/docs/instrumenting/exporters/.

31. P, Sudheesh. *A DEEP DIVE INTO LOG MONITORING USING ELASTIC STACK* [online]. [visited on 2023-03-24]. Available from: https://blog.qburst.com/2020/01/a-deep-dive-into-log-monitoring-using-elastic-stack.

32. *Elasticsearch on Kubernetes: A new chapter begins* [online]. [visited on 2023-03-25]. Available from: https://www.elastic.co/blog/introducing-elastic-cloud-on-kubernetes-the-elasticsearch-operator-and-beyond?elektra=products&storm=sub1.

33. *Install Elastic Agents* [online]. [visited on 2023-03-29]. Available from: https://www.elastic.co/guide/en/fleet/8.6/elastic-agent-installation.html#elastic-agent-installation.

34. *Cloud Native Computing Foundation announces Prometheus graduations* [online]. [visited on 2023-04-02]. Available from: https://www.cncf.io/announcements/2018/08/09/prometheus-graduates/.

35. *Doubling down on open, Part II* [online]. [visited on 2023-04-02]. Available from: https://www.elastic.co/blog/licensing-change.

36. *APM register custom metric* [online]. [visited on 2023-04-02]. Available from: https://www.elastic.co/guide/en/apm/agent/nodejs/current/agent-api.html#apm-register-custom-metrics.

37. *Data in: documents and indices* [online]. [visited on 2023-04-02]. Available from: https://www.elastic.co/guide/en/elasticsearch/reference/current/documents-indices.html#documents-indices.

38. CARCASSI, Gabriele et al. SLATE: Monitoring Distributed Kubernetes Clusters. In: *Practice and Experience in Advanced Research Computing*. Portland, OR, USA: Association for Computing Machinery, 2020, pp. 19–25. PEARC '20. ISBN 9781450366892. Available from DOI: 10.1145/3311790.3401777.

39. *Default limits applied by the operator* [online]. [visited on 2023-04-03]. Available from: https://www.elastic.co/guide/en/cloud-on-k8s/current/k8s-managing-compute-resources.html#k8s-default-behavior.

40. *kube-prometheus/prometheus-prometheus.yaml at main · prometheus-operator/kube-prometheus* [online]. [visited on 2023-04-03]. Available from: https://github.com/prometheus-operator/kube-p

rometheus/blob/main/manifests/prometheus-prometheus.yaml.

41. *helm-charts/charts at main · grafana/helm-charts* [online]. [visited on 2023-04-03]. Available from: https://github.com/grafana/helm-charts/tree/main/charts.

42. [*Security Solution*] *Endpoint Security Integration fail with Fleet Managed Elastic Agent · Issue #137749 · elastic/kibana* [online]. [visited on 2023-04-04]. Available from: https://github.com/elastic/kibana/issues/137749.

43. *k3s-io/k3s: Lightweight Kubernetes* [online]. [visited on 2023-04-13]. Available from: https://github.com/k3s-io/k3s/.

44. *kubernetes/kube-state-metrics: Add-on agent to generate and expose cluster-level metrics.* [online]. [visited on 2023-04-11]. Available from: https://github.com/kubernetes/kube-state-metrics.

45. *curl* [online]. [visited on 2023-05-14]. Available from: https://curl.se/.

46. *Logs reference* [online]. [visited on 2023-05-03]. Available from: https://docs.elastic.co/en/integrations/system#system.

47. *Scalability and resilience: clusters, nodes, and shards* [online]. [visited on 2023-05-04]. Available from: https://www.elastic.co/guide/en/elasticsearch/reference/current/scalability.html.

48. *Advanced configuration* [online]. [visited on 2023-05-05]. Available from: https://www.elastic.co/guide/en/elasticsearch/reference/current/advanced-configuration.html#advanced-configuration.

49. *HostPath* [online]. [visited on 2023-05-06]. Available from: https://kubernetes.io/docs/concepts/storage/volumes/#hostpath.

A Electronic Attachments

The thesis archive contains a ZIP archive `attachment.zip`. All the source code and configuration files necessary to run the proof of concept are in this archive.

The files in the archive have the following structure:

- `README.md` – Markdown file with instructions on how to deploy the proof of concept and its hardware and software requirements,
- `targets` – directory containing the source code and configuration files for the sample applications,
- `scripts` – directory with bash and python scripts to simplify the deployment,
- `monitoring` – directory containing YAML configuration files to deploy ECK, Elastic Stack and kube-state-metrics,
- `dashboards` – directory with custom Kibana dashboards.