University of Zurich^UZH

Master Thesis July 19, 2021

Small teams, complex platforms

Analysis and improvement of maintainability for the Beelivingsensor cloud platform

Vladimir Masarik
of Zurich ZH, Switzerland

Student-ID: 19-761-766
vladimir.masarik@uzh.ch

Advisor: **Dr. Clemens Mader**

Prof. Dr. Lorenz Hilty
Department of Informatics:
Information and Sustainability
Research Group
University of Zurich
https://www.ifi.uzh.ch/isr

Acknowledgements

Without a doubt the biggest gratitude goes to Daniel Boschung, Dr. Clemens Mader, and Prof. Dr. Lorenz Hilty. The whole project could not work without their support, and the opportunity they gave me to have this thesis. I would have never thought that I would get to work on such a fitting topic as this one. I am especially thankful for their understanding, and patience with me solving all the problems.

Incredible thanks goes to Anna Jancso, as she was always available to provide me technical guidance, and to all the previous researchers on the Beelivingsensor project. Same goes for a great friend Severin Siffert who provided me with mental support, and phenomenal tips on how to make the writing part of my work manageable. Lastly, my thanks goes to Florian Ruynat, Denis McDermott, and the whole FOSS community. Such amazing people, and sincerely my idols who provided me with crucial technical guidance. I hope to become a FOSS contributor myself, and people like them represent my endless motivation.

Abstract

Estimating biodiversity of a local environment can help us decrease the rate of its degradation. The Beelivingsensor project aims to utilize honey bees as *living sensors* that could help us estimate biodiversity. However, in order to analyze the honey bees, and therefore, extract the valuable information, we need to process a lot of data. This need is fulfilled by the Beelivingsensor platform, as it accepts data from multiple sources, and processes them to provide us with useful information.

Our thesis aims to improve the maintainability, robustness, and extensibility of this platform. Our hope is that we will help developers focus on improving the platform, rather than making sure it is operational. Moreover, we want to make sure that in case no experienced developers are available, and our platform encounters errors that could endanger its functionality, the platform will stay operational, and fixing any problems will be as easy as possible.

As a result of our research, we were able to identify five common problems affecting cloud platforms according to the literature. We have then used this knowledge to improve the maintainability of our platform in eight key areas. Additionally, by conducting two experiments, we were able to objectively prove that outside of our own positive experience, these improvements are indeed beneficial. During the experiments, we have also noted a few unexpected observations, which could be further explored in the future research.

In the end, we have helped the platform to stay operational for longer periods of time, and we have decreased the amount of attention it needs from its administrators. The research can also help other Kubernetes platform maintainers to reconsider the tools they should prioritize, and they might be able to avoid some pitfalls that we have discovered during our experiments.

Zusammenfassung

Die Einschätzung der Biodiversität einer lokalen Umgebung kann uns helfen deren Verschlechterung zu verringern. Das Projekt Beelivingsensor zielt darauf ab, Honigbienen als *lebende Sensoren* zu nutzen, die uns helfen könnten, die Biodiversität abzuschätzen. Um die Honigbienen zu analysieren und damit die wertvollen Informationen zu gewinnen, müssen wir jedoch viele Daten verarbeiten. Diesem Bedarf wird die Plattform Beelivingsensor gerecht, da sie Daten aus mehreren Quellen entgegennimmt und diese verarbeitet, um uns nützliche Informationen zu liefern.

Unsere Arbeit zielt darauf ab, die Wartbarkeit, Robustheit und Erweiterbarkeit dieser Plattform zu verbessern. Wir hoffen, dass wir Entwicklern dabei helfen, sich auf die Verbesserung der Plattform zu konzentrieren, anstatt sicherzustellen, dass sie betriebsbereit ist. Darüber hinaus möchten wir sicherstellen, dass für den Fall, dass keine erfahrenen Entwickler verfügbar sind und unsere Plattform auf Fehler stößt, die ihre Funktionalität gefährden könnten, die Plattform betriebsbereit bleibt und die Behebung von Problemen so einfach wie möglich ist.

Als Ergebnis unserer Recherche konnten wir laut Literatur fünf häufige Probleme identifizieren, die Cloud Plattformen betreffen. Dieses Wissen haben wir dann genutzt, um die Wartbarkeit unserer Plattform in acht Schlüsselbereichen zu verbessern. Darüber hinaus konnten wir durch zwei Experimente objektiv nachweisen, dass diese Verbesserungen außerhalb unserer eigenen positiven Erfahrungen tatsächlich von Vorteil sind. Während der Experimente haben wir auch einige unerwartete Beobachtungen festgestellt, die in der zukünftigen Forschung weiter untersucht werden könnten.

Letztendlich haben wir dazu beigetragen, dass die Plattform über einen längeren Zeitraum betriebsbereit bleibt, und wir haben die Aufmerksamkeit der Administratoren verringert, die sie benötigt. Die Forschung kann auch anderen Betreibern von Kubernetes Plattformen helfen, die Tools zu überdenken, die sie priorisieren sollten, und sie sind in der Lage, einige Fallstricke zu vermeiden, die wir während unserer Experimente entdeckt haben.

Contents

1

Introduction

This chapter describes the current situation and context in which we are moving. Section 1.1 establishes the context, and project background of the Beelivingsensor platform, which we are trying to improve. Section 1.2 then defines the problems we are dealing with, and trying to solve, while also presenting specific research questions which we address in this thesis. Lastly, section 1.3 explains in detail any technologies and concepts that we are extensively using in our thesis, so that the readers can better understand them.

To further introduce the structure of this paper, the *Common Issues* chapter 2, presents issues that cloud developers have to frequently deal with. The chapter aims to help us better answer one of our research questions (RQs). Concrete answers are then provided in the chapter 3, where each section explores an area where we were able to improve the maintainability of our platform

Afterwards, there is an *Evaluation* chapter 4, in which we present two experiments. There, we want to show that the improvements that we have implemented, are objectively beneficial, and the results are replicable. Last is the *Discussion* chapter 5, where we discuss various limitations, future work, and other observations that we believe are worth mentioning.

1.1 Project Background

With modernization and increasing urbanization, the biodiversity of many areas is decreasing. The natural cycles are being unknowingly disrupted, and in many cases we are not able to fully understand how these disruptions damage environments. We can however say that loss of biodiversity is not helping the stability of various ecosystems.

As such, there are initiatives that try to restore biodiversity, or at least slow down the process of its degradation. However, unless they are able to measure their progress, we are not able to say which initiatives are worth pursuing, and therefore viable in the long term. Of course, we are able to measure the biodiversity of the local environments to some degree, but many of these techniques require substantial financial and human resources. Therefore, many of the techniques are not able to cover larger areas.

The Beelivingsensor project aims to solve exactly this issue, as honey bees are great pollinators that spread over large areas, and collect pollen from many sources. The project's goal is to use this fact, and therefore use the bees as *living sensors* capable of measuring the local biodiversity, hence the project name Beelivingsensor. Additionally, we want to use not only the bees, but also other data sources to predict the local biodiversity.

Still, the major source of information should be the bees, and we would like to extract data about their activities and behavior using video recordings of their beehives. Specifically, we focus on the pollen color and bee traffic at the beehive entrance. Our project is also special in the sense that we hope to monitor the bees using non-invasive methods. This should mean that we do not disturb the natural behavior of the bees, and that no special equipment has to be manufactured in order to successfully extract the data, making our technique more financially viable.

Moreover, we hope to help the beekeepers as well. Although our project aims to estimate the local biodiversity, and the research community with support from governments should be motivated enough to implement our solutions on a larger scale, we believe that we can further help with the adoption rate of our solution by providing benefits to beekeepers as well. We believe that the beekeepers can benefit from taking part in our project by being able to monitor health and activity of their bee colonies. Therefore, they should be able to better plan ahead with their feeding schedules, or they should be able to tell much sooner when their colony's health is degrading, and it needs more attention.

The Beelivingsensor project aims to provide these benefits by creating a platform where the researchers can collect information about the biodiversity of various areas, while the beekeepers can keep monitoring their colonies. We can achieve this by collecting information from various sources, such as the beehive videos, local plant populations, or from sensors placed near beehives.

In order to complete the objectives of the Beelivingsensor project, a cloud platform[1] was created. The platform is a concrete implementation of every goal that our project aims to achieve. Meaning that it allows the beekeepers, researchers, and citizen scientists to register, and use its features as necessary.

The beekeepers should be capable of creating beehives on our platform, which represent their own beehives in real life. They are then able to observe various collected metrics for each beehive, but also the whole colonies. From there, they are able to upload recording of each beehive either manually using their web browser, or through File Transfer Protocol[2] (FTP), which can be used automatically by the cameras that record the beehives. The videos are then automatically analyzed, and the beekeepers can view any extracted information as well.

We also integrate with various other external APIs from different organizations. These APIs provide us with further information, which we can use to estimate the local biodiversity, but also health of the bee colonies. Therefore, we help the beekeepers and the researchers to gain insights by aggregating multiple data sources into a single website, and they do not have to visit multiple places in order to create relations between the

[1]https://www.beelivingsensor.eu/en/
[2]*https://en.wikipedia.org/wiki/File_Transfer_Protocol*

different data sources. For example, we are able to provide the information about the pollen color that bees bring into the hive, while also automatically showing references to what plants have been detected in the area around the beehives.

Anyone registered on our website is also capable of improving our machine learning models that analyze the videos. Therefore, anyone interested is able to verify and adjust labels that are created by our models, and these adjustments are afterwards used to improve our machine learning models.

1.2 Problem Description

As with any other projects, we hope to grow and improve our functionality in the future. This of course requires developers and scientists that help us design new functionality, and eventually implement it. The developers are however needed for maintenance and operation activities as well. We need people to make sure that if for some reason a part of our platform stops suddenly working, we will be able to quickly recover our operations. Considering that at the time of writing the platform has been in active development for the last year, the source code is still relatively new, and could be prone to errors. The reason being that during the last year, we have focused on new functionality rather than stability.

With many parts of the platform being functional however, we need to improve the robustness and stability of the platform. We need to make sure that if something fails, it will not cause the whole platform to be dysfunctional. We also need to decrease the chance of errors happening, while improving the debugging capabilities of developers, so that if an issue happens, we will be able to quickly fix it.

Additionally, we want the developers to spend as little time as possible on maintenance, or any other frequent events, so that they can focus on improving the platform rather than keeping it functional. We also believe that maintenance requires a different set of skills, which would require the developers to learn them, thus, it would slow down developers even further.

In the end, we can summarize the goals of this thesis as:

- Explore, analyze, and implement solutions that ensure stability and robustness of the platform.

- Explore, analyze, and implement solutions that help developers extend and maintain our platform.

We are also able to rephrase these goals into specific research questions (RQs), which are the following:

- RQ 1.) What are the possible ways of decreasing the maintenance efforts?

  - RQ 1.1.) How can we simplify the process of finding root causes of errors for developers?

  - RQ 1.2.) What are the possible ways of decreasing platform administrator response times in case of abnormal events?
  - RQ 1.3.) How can the platform stay functional even in case of unexpected application crashes?
  - RQ 1.4.) What other events are platform administrators expected to handle, and how can we automate their handling?

- RQ 2.) Which of the identified solutions of each RQ 1. sub-questions are beneficial to implement for the Beelivingsensor platform?
- RQ 3.) What are the benefits gained from each implemented solution?

1.3 Fundamentals and Terminology

This section aims to help the reader orientate themselves in the concepts used throughout the text. We hope to provide more information about the various parts of our platform, and what software is needed for it to function. We also hope to provide a much deeper understanding of software that was necessary for a successful implementation of the solutions that helped us improve the maintainability of our platform. In order to fully understand our explanations, the reader is expected to have at least a basic familiarity with GNU/Linux based OS, and common CLI tools that are provided by the OS.

Docker. The naming conventions around Docker are a bit confusing, so we would like to clear them out before explaining what Docker is. Currently, there is a software project called Moby[3], and this Moby project is used as the main source of functionality for the *Docker Community Edition*[4] (Docker CE). Therefore, any functionality that the Moby project provides, is included in the Docker CE. The Docker CE however, also contains some extra features unique to Docker CE. Throughout this thesis, unless stated otherwise, we refer to Docker CE simply as *Docker*.

The reasons behind the naming complexity are historical. When the Moby project was released, it was at first simply called Docker. With time however, the Docker project gained a lot of popularity, and considering that the company that released Docker was also called Docker, it created a lot of confusion. Therefore, the company decided to extract the defining features into a standalone, free, and open source project called Moby. In the meantime, the company continued to maintain the Docker CE so that users, who were happy with the Docker, do not have to switch to Moby. Additionally, the Docker company released a commercial version of Docker CE referred to as *Docker Enterprise Edition.*

Finally, we can talk about the Docker functionality. It is an OS-level virtualization platform, on which you can manage your applications[17]. Meaning, in a very similar way to virtual machines, Docker allows us to create isolated and identical environments,

[3]https://github.com/moby/moby
[4]https://www.docker.com/

in which we are able to run our applications. Therefore, it is possible to create packages, which are called *Docker Images*, that allow us to execute our applications, irrespective of the OS on which the Docker is installed.

When developers are creating an application, they have to write a source code that defines what the application does. The source code can then be compiled, and the outcome is a binary. If the application is also using some external libraries, and it is dynamically linked, the binary requires those external libraries in order to function properly when it is executed.

In traditional deployments, the applications are in the same environment as other applications, and each one has access to the same libraries, as is portrayed in Figure 1.1. When working with Docker however, the developers create a configuration file often referred to as *Dockerfile*. This file specifies various attributes of a Docker image, but most importantly it specifies the contents of the Docker image. As we have mentioned, the Docker image is basically a package, and the developers include in this package their application, and anything else that is required to successfully execute it, e.g. the external libraries. The Docker image is created by Docker using a *build* command, and in very simplified terms, the outcome is a file representing the Docker image, which holds the application and its dependencies.

This Docker image can then be executed using the Docker, which results in Docker running the application contained in the Docker image. In a similar fashion to virtual machines (VMs), where you can create a VM from its image, the Docker does not consume the Docker image, and instead it creates a virtual environment containing the application and its dependencies. This environment is referred to as a *Docker container*. Docker can create many containers, which are basically all clones of the Docker image, and they run irrespective of each other. Compared to the traditional deployment, when using Docker, each application has its own environment and necessary libraries, as can be seen in Figure 1.2.

Kubernetes. Docker containers share many benefits with VMs, but they are smaller, and faster to start. There is a problem though, which is that you can start and stop the containers, but the users are not able to manage them in any other useful ways. This is a goal of Kubernetes however, which is a platform for containers that allows the users to manage them. Kubernetes provides a lot of advantages, but specifically, it allows the users to manage configuration of the containers, it simplifies networking between them, scaling of your applications, and their storage management[23, 34].

In order to fully use Kubernetes, the developers have to create a *Kubernetes cluster*. This is done by creating VMs, then installing Kubernetes on them, and letting each Kubernetes installation know about the other installations. In the end, the users will have a few VMs with Kubernetes installed on them, and these installations will be communicating between each other. Together, these Kubernetes instances create a Kubernetes cluster, and having that cluster allows the abstraction where you do not run your applications on a VM, but on the Kubernetes platform. Technically, the applications are still running on individual VMs, but you as a user do not mind which VM they are running on, you simply instruct Kubernetes to execute your application.

Kubernetes has a few concepts that are unique to it, and the most basic concept

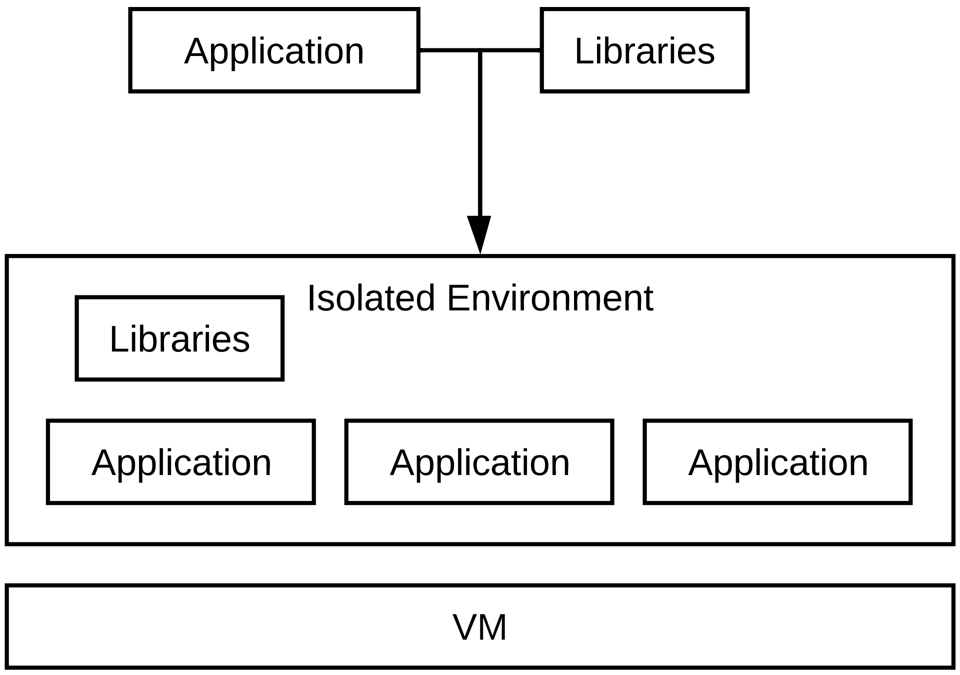


Figure 1.1: Representation of traditional application deployment environments.

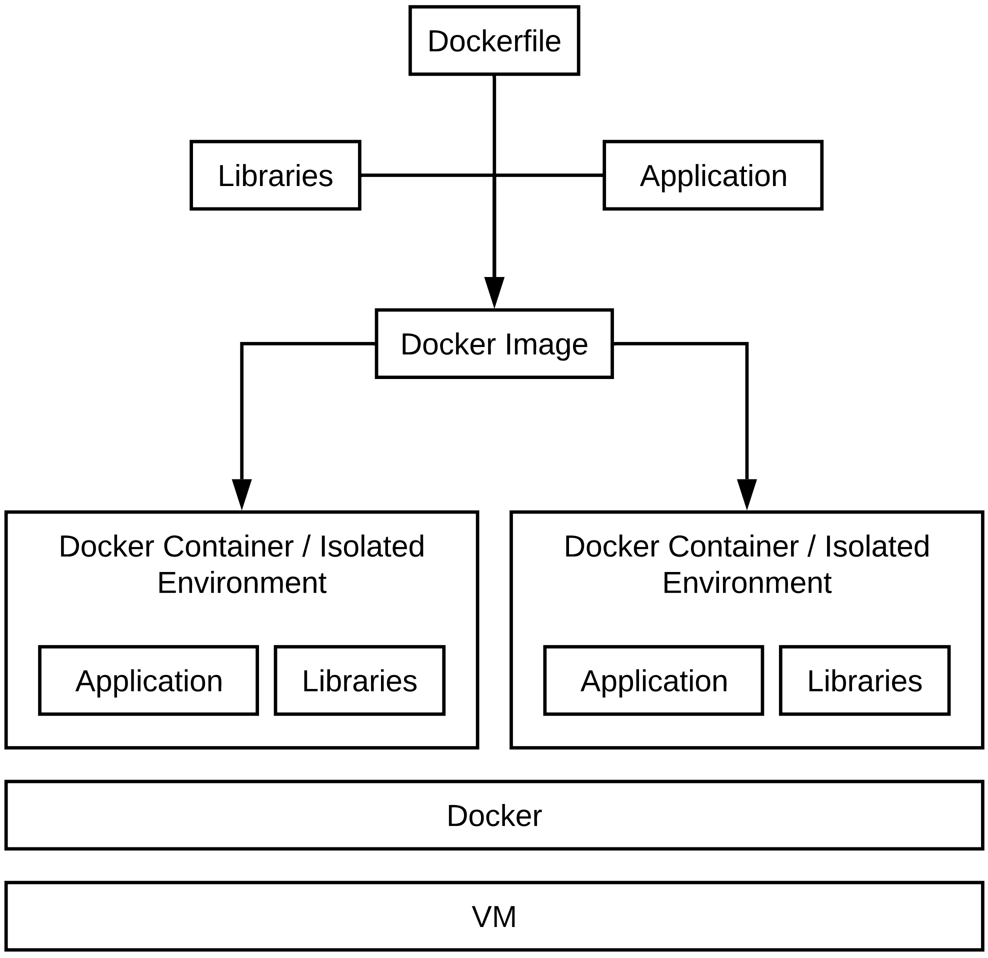


Figure 1.2: Representation of application deployment environment using Docker.

is a *Pod.* Simply put, a pod is a logical group of containers, which are created and deleted together. Very often, pods contain only one container, and therefore only one application, which results in people using these concepts interchangeably.

There is also a higher level representation of a pod, which is referred to as *Deployment.* They are supposed to represent a single micro service or an application. Pods are concrete instances of the application, but these instances can be deleted at any point without a warning. Deployment then represents every identical instance running on top of Kubernetes. Therefore, it allows you to define how many clones of your application should be running at any point, and any other information that uniquely defines a micro service.

Users can create representations of their applications on Kubernetes, such as pods and deployments. There are however cases when it would be useful to create new Kubernetes concepts, and therefore extend the functionality of Kubernetes. This functionality is allowed by *Custom Resource Definitions* (CRDs). They allow developers to specify custom functionality that Kubernetes did not have before. For example, there is an SSL certificate CRD[5] which enables the users to represent SSL certificates in Kubernetes, and build functionality on top them. You can see a representation of all these concepts in Figure 1.3.

Kubernetes, as an application, exposes an HTTP API with which the users communicate, in order to manage their applications. Users can of course send their requests directly to the API, but Kubernetes has also a CLI tool *kubeclt*[6] that creates these request. Therefore, users do not have to fully understand the API format, and can use the *kubectl* to issue commands that can create, delete, or adjust their Kubernetes resources, e.g. "*kubectl create pod my-app ...*".

Infrastructure. Concerning the creation of the Kubernetes cluster, we would like to briefly outline how it is done. Since we are using Azure[7] cloud provider, the physical infrastructure is maintained by Microsoft. Azure however exposes an API, which allows us to request new Azure resources, such as disks, load balancers, virtualized networks or VMs. Terraform[8] is able to communicate with this API, and it allows us to define which infrastructure resources will be created. In other words, we can write so-called *Terraform files*, which define virtual networks, VM sizes, VM names, disk sizes, and anything else that Azure can provide us with. Then, we can provide Terraform with these files, and it will communicate with the Azure API in order to deploy our resources. If everything succeeds, we are always able to deploy identical infrastructure, onto which we can then deploy Kubernetes.

The setting up of a Kubernetes cluster is not trivial. There are many details depending on which cloud provider you are using that you have to configure properly. Kubespray[9] is a community project supposed to automate the Kubernetes cluster deployment process as much as possible, which we also use to deploy our cluster. We just need to provide

[5]https://github.com/jetstack/cert-manager/blob/master/deploy/crds/crd-certificates.yaml
[6]https://kubernetes.io/docs/reference/kubectl/overview/
[7]https://azure.microsoft.com/en-gb/
[8]https://www.terraform.io/
[9]https://kubespray.io/

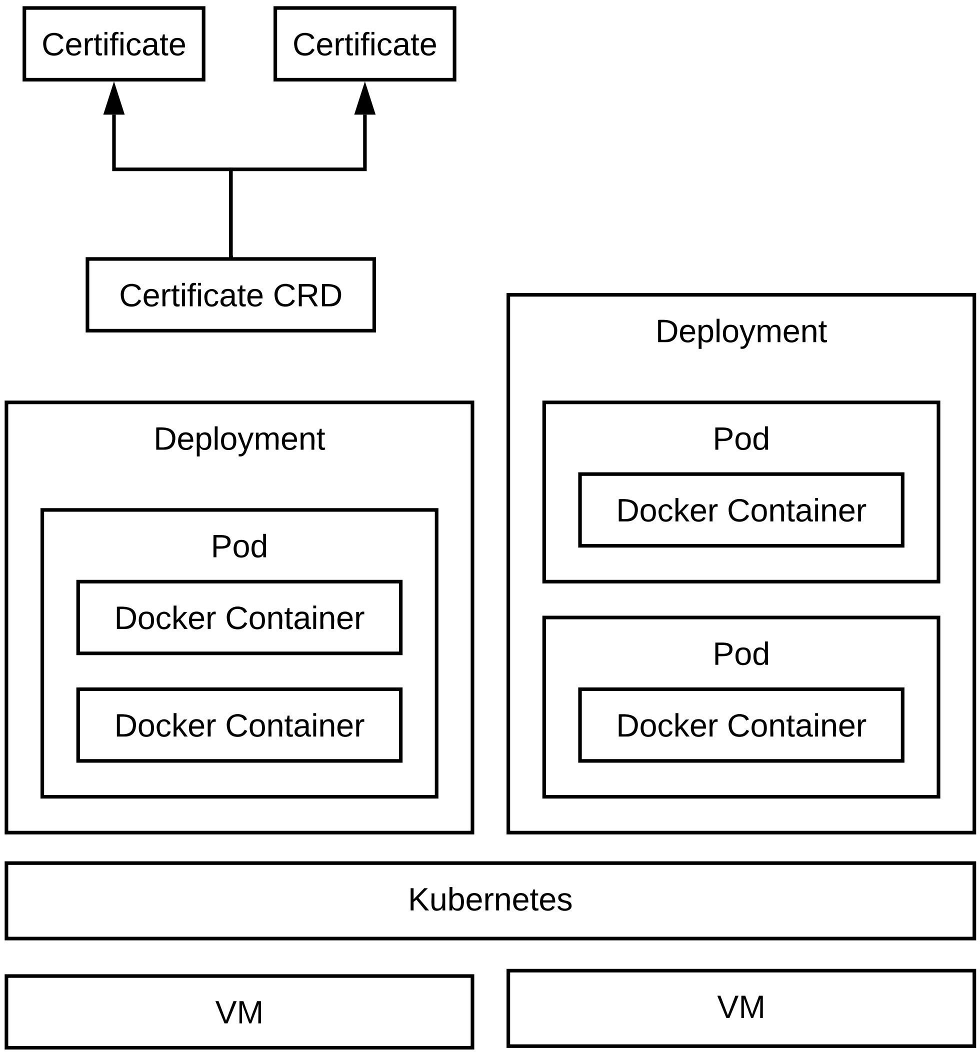


Figure 1.3: Kubernetes environment and its concepts.

connection details to the VMs that should represent the Kubernetes cluster, and adjust the deployment configuration based on our situation. After the deployment is successful, we are able to use Kubernetes to host our applications, and people can start using them.

Many of the tools, that we have implemented as part of this thesis, are using Helm[10] in order to deploy them on Kubernetes. Helm can be described as a package manager for Kubernetes. Meaning, we are able to specify an application that is meant to be run on Kubernetes, and we are able to easily deploy it. As with any other package managers, it automates a large part of the deployment process, although it still requires us to provide it with a few configuration options that instruct the applications to behave as we want them to. If Helm does not receive custom values when deploying an application, it will use default values, which in many cases result in successful deployment, but the application does not work as we might need it to. Therefore, finding the correct configuration values is the core problem when deploying any Kubernetes applications.

[10]https://helm.sh/

2

Common issues

In this chapter we will describe common problems or errors that can happen during the development and operation of a cloud platform. That includes mainly the infrastructure and platform itself, but also the applications that are hosted on it in general.

The goal of this chapter is to help us answer the RQ 1.1.. We want to know how we can simplify the process of finding root causes of errors for developers, but the question is somewhat too broad to be answered in a useful way. Therefore, we want to first find what are the most common problems that occur in cloud platforms such as ours. Then, we can also analyze the problems, and find out how we could simplify the debugging process for the developers. Later, in chapter 3. we present concrete solutions that would help us with debugging these common problems.

The reason for finding common problems first is that focusing on the most frequent problems, means that improving our debugging capabilities in such situations would mean that our maintainability will be improved the most. Additionally, by focusing on a smaller set of problems we are able to suggest better solutions, compared to if we had focused on every problem in general. Therefore, we hope to come with useful solutions that are able to improve our maintenance the most.

Moreover, for the last year, developers of the Beelivingsensor platform focused on bringing new functionality, but not necessarily on improving the maintainability. Therefore, we expect that there are many easy to implement, and yet highly beneficial ways of improving it.

Additionally, in the future we do not know what other applications might be hosted on our platform. Over time, the applications might change, and new bugs will be introduced, but the way these problems will be debugged will stay the same. We are not able to prevent future issues from happening. They might have different root causes, and covering the current bugs, might not be as helpful in the long term. Thus, allowing current and future developers to better debug their problems will speed up their development in general, and we will avoid solving problems that might become irrelevant later on.

Lastly, it is possible to stop the development of the applications that are running on Kubernetes. It is possible that the platform will reach its desired state, and there would be no more need to expand it. At that point, we would expect that the applications would be rather stable, and the probability of introducing new bugs would be minimal, as there would be no new changes introduced for the applications. However, the platform will be still running, and it will still require maintenance. Therefore, if we are able to decrease the maintenance required for keeping the platform operational, we believe that we will also prolong the lifetime of the platform, as the developers will be less discouraged to keep maintaining the platform, as it will require less effort.

In order to find the most common issues happening in cloud platforms we have looked at various research papers. Specifically, we have mainly used research of Guwani et. al. and Liu et. al. to identify the common problems[44, 45, 47]. There are a few caveats though, as the Guwani et. al. did not look at *common problems*, but rather bugs that occur in distributed cloud systems[44]. Moreover, their research from 2016 overlaps in some of the issues they found in 2015, but they are looking at a different type of problems in the cloud[45]. Furthermore, Liu et. al. provided us with valuable data as well, however, they have identified bugs that can be mostly included in the misconfiguration or application bugs categories[47].

We have also considered research of Chen et. al., as they look at bugs such as incorrect error handling. They classify bugs based on what triggered them, and they divide the causes into semantic and non-semantic[38]. The non-semantic errors are then further subdivided into issues related to network, file system, lack of computing resources, and interruptions. Unfortunately, they do not specify exactly what is included in the semantic causes, however, they do mention that it is the errors caused when "*variables are set to an incorrect value*", which we can interpret as misconfiguration issues.

As you may have noticed, almost none of the researchers provide a list of similar categories that other researchers would use as well. Thus, unfortunately for us, it seems that the research is rather uncertain about which are the common problems affecting cloud applications[45, 36, 40]. This proved to be rather difficult for us, as finding a definitive answer on what are the most frequent issues in the cloud seemed impossible.

We have noticed that this might be caused by the fact that the research papers do not seem to have a unified view on constitutes a bug, as it is highly dependent on the context of the research paper. As an example, Guwani et. al. in their 2016 claims that bugs are, alongside misconfiguration, one of the root causes of cloud incidents[45]. While their 2015 research lists misconfiguration as part one of the software bugs[44]. Therefore, we are not able to clearly say what are the most common issues affecting cloud platforms, and its applications.

In the end however, we were able to note five of the most common problems that were usually repeating in every research paper that we have analyzed. Specifically, we looked at issues with network, application bugs, misconfiguration, performance or traffic overload, and external services, all of which are further described in their respective sections.

In order to provide solutions for these common issues, we have explored various sources. Specifically, we looked at different open source projects hosted on GitHub[1] and GitLab[2], we have also explored Stack Overflow[3] answers for suggestions, multiple

[1]https://github.com/explore
[2]https://gitlab.com/explore
[3]https://stackoverflow.com/tags

blog post written by individuals, but also companies sharing how their platforms work. Moreover, we have looked at conference talks explaining how they approach common problems on their platform, and lastly, we have looked at various Cloud Native Computing Foundation[4] (CNCF) projects that could help us.

2.1 Network Failures

Kuberenetes, and the applications it manages, create a network, which is a complex and distributed system where the individual applications communicate between each other. To function properly, the network has to provide ways of finding the other services, but also make sure that the HTTP requests have reached its target. To make sure that all of these requirements are met, Kubernetes allows the users to install network plugins commonly referred to as *Container Network Interface* (CNI). The CNIs play a crucial part in Kubernetes networking, and it is one of the Kubernetes concepts that developers often meet with, especially in the area of networking because most of the other details are hidden behind the CNI configuration.

Traditionally, when a VM is created, it has network interfaces, such as *lo* for loopback or *eth0* for ethernet. However, when we want to host multiple applications on a single VM using Kubernetes, and if all of the applications want to use the same HTTPS port 443 for serving the user requests, it results in conflicting ports.

Kubernetes and CNI solve this problem, and allow the applications to occupy the same ports on the same VM. CNI introduces virtual network interfaces, which then help with management of the applications, and they allow us to run multiple applications on the same VM without conflicts[3, 46]. Of course port collisions are not the only problem that CNIs solve, but it was a great example to describe the area in which they help, without going into too many details of what they do.

Moreover, Kubernetes is still a relatively new technology, as it was introduced in 2014. Therefore, the CNI plugins are even newer concepts, and can contain bugs. Many of the CNI plugins do not fully support the three largest cloud providers either. Even we experienced problems with CNI because the default CNI plugin Calico did not support Kubernetes on Azure, although it was added rather recently[4].

As such, networking is a large part of Kubernetes, although it is not necessarily perfect yet. Therefore, the users will have to eventually work with network configuration at some point. Although, many times the bugs are responsible for the network errors, misconfiguration of the network is the second most common cause of issues with the network[45]. Specifically, the most impactful problems are when the services either do not know about each other, or cannot reach each other.

Therefore, in order to ensure a stable platform we need to have tools that would allow us to catch, and investigate errors as quickly as possible. Otherwise, if our applications would stop communicating, most of the functionality that our platform provides would be unavailable.

[4]https://landscape.cncf.io/

In our current deployment, we fortunately do not have any complex networking needs. Our platform focuses on serving a simple website and a few APIs, and we are not running a customized DNS service, or anything else that requires a lot of network configuration. So far we are not using any technologies that would modify our network in a considerable way such as queues or other traffic managers either.

Moreover, traffic inside our cluster is using HTTP and not HTTPS. The incoming user requests are using HTTPS, but after reaching our load balancer, the traffic continues to the applications via HTTP. Thus, we are able to avoid problems with authentication and certificate management between our applications.

We also have a rather simple network architecture in terms of how our applications communicate. Not only do we have just a few core applications, but after the Internet traffic arrives to the load balancer, it is mostly forwarded to our web application. From there the application communicates with other services as necessary, which is mostly our video analysis application. Thus, most of our network is rather linear, and does not diverge too much.

In terms of how frequent we can expect the network problems to be, we would expect them to be rather rare. Considering that most of the networking problems occur during upgrades, and as a result of misconfiguration, we believe we are somewhat safe. Specifically, there are not many cases in which we modify, configure or in any way adjust the network, after the initial creation of the platform, nor do we plan to do so. The only events where we would expect the network to fail is during upgrades, and so far we plan less than two upgrades per year, during which we also upgrade the Kubernetes versions.

We still would not like to underestimate the situation. Therefore, we believe there are still many ways in which we can improve the debugging capabilities of networks. One of the most helpful tools would be tracing. It is capable of creating graphs of how our HTTP requests travel across our cluster. These traces can be thought of as stack traces for processes, which show where each request was sent to, and how long it spent being processed in which application. This can easily help us debug traffic in a very granular way, and we can also create topology maps of our platform. Thus, we are able to clearly understand how our requests are being routed, and where the problem could be.

Another concept that could help us with debugging would be collecting network metrics. Using these metrics, we should be able to quickly understand the overall situation, and which areas need our attention. For example, we would be able to see which services are failing, and what are the error codes that are being raised. Afterwards, we can dive deeper into the exact details of what is causing the problems.

In the process of uncovering the issues in detail, we could use a centralized log collection system. Therefore, when we would be identifying where the issues are coming from, we would be also capable of quickly querying the logs of relevant applications. In other words, the log collection would help us collect logs from the applications, and then allow us to search them. As such, we expect we would be able to quickly find the exact time of when the errors started appearing, while also filtering out irrelevant logs. Thus, it would further help us in understanding the core issue causing network problems, as we would know the exact details of why the applications are failing.

2.2 Application Bugs

Naturally, bugs are an unwanted part of every software, and our applications are no different. We would like to have software that is free of bugs, but nobody writes perfect code, and we will have to eventually debug a few of them. Especially, given that we create software that is supposed to work in a complex distributed environment that is frequently changing. Therefore, the less time we have to spend on finding the root causes the more time we will have to develop new features, and our platform will be easier to maintain, as we will cut down the time it takes to find an error.

Debugging problems in cloud environments is not necessarily easy. Kubernetes is used to host distributed applications, where the applications are encapsulated in Docker containers. This does not help with debugging, as there are no easy ways to attach debuggers to the applications. Moreover, Kubernetes helps us in making sure that our development environment is identical to the production environment, however there are still a few differences that make it difficult to replicate the application states, such as different data that is being processed.

These difficulties then complicate debugging of the applications. Specifically, it is hard for us to know what is or what was the exact state of the application when the error occurred. The application can express the relevant information into the logs, or in some other ways, but we also need to make sure that this information can easily reach the developers. Thus, printing logs is only one of the parts necessary to fully understanding what is happening within the application.

Currently, the most common way of debugging the applications is that we create a *bash* session within the pod itself, and then using different commands, we try to better understand the situation. Unfortunately, we are somewhat limited in what we can do, and how we can interact with the application. We usually also print the logs of the application using the Kubernetes CLI tool *kubectl*. However, practical usage of the logs is somewhat limited, as the applications create a lot of logs, and it is often hard to search through them.

Moreover, our usage of logs is restricted in a few ways. We are able to search logs of only one application at the time, which means that more complex bugs would be fairly difficult to solve, as we would not be able to easily cross reference the different errors happening in different applications. Additionally, browsing logs of pods that have been deleted is difficult in terms of finding and accessing them, but also by usually having large amounts of logs that we would have to search through.

With regards to frequency of the bugs, they will always be present in one way or another, as they are a natural result of an evolving software. They are the third most common cause of cloud incidents, and there is a lot of research being done on how we can decrease the amount of bugs[44, 45]. Therefore, we expect to encounter them very often, which is also why we want to make sure that we are able to solve them quickly.

Additionally, there are going to be more and less critical bugs present in our applications. Of course, we are going to be solving the most crucial problems first. Thus, improving our investigative abilities would especially be helpful for the more crucial bugs, as we would be focusing on them most of the time.

Specifically, we would like to talk about three applications that we believe are the most crucial on our platform. We believe that if we are able to improve the debugging speeds of these applications, then the rest of applications that we run should be improved as well.

The first application would be the model analysis. It is responsible for processing videos using available machine learning models, and extracting information out of them. The information is then provided to our web application, which further processes them. As you can imagine, the model analysis holds a lot of information about the videos it uses. Moreover, the application is very crucial to the functioning of our platform, so every piece of information that we can extract from it, can help us debug its problems.

For example, since the application uses GPUs to process the videos, knowing the usage of the GPU is rather important to understanding how the application behaves. This information can be however only obtained if we have access to the VM where the model analysis runs. Thus, it slows down the speed of debugging, and can be impossible to obtain in certain situations where the VM becomes inaccessible. Moreover, the application creates a few GB of logs per day, which are of course not all relevant, and therefore, finding the useful logs becomes difficult. It becomes especially difficult if the application is running for multiple days, and we have to obtain all of the logs, except we do not necessarily need all of them.

The second application we would like to mention is our web application. As one could assume, it is responsible for serving our website to the users, but it also holds most of the logic which decides how the videos are processed, and how the videos are saved in our platform. Therefore, it is critical for us to fix any problems that we might experience as soon as possible.

Similar to the model analysis application, our web generates a lot of logs, as it faces the Internet, and anyone can create a request to it. As a result, logs contain many irrelevant lines because web crawlers and other botnets are trying to scan it for vulnerabilities. Unfortunately, logs are the most useful source of information about the internal state of the application. Therefore, being able to filter out logs that are not relevant for us, and finding the exact information we are looking for, is the most helpful ability we could use. At this time however, we do not have any concrete tools that would help us in such ways.

Lastly, we try to write stateless applications, which can be easily scaled and managed by the Kubernetes. However, the data still has to be held somewhere, which is our SQL database. There are quite a few cases in which bugs happen only when using certain data or in certain states. Thus, it is useful to know what exactly is the state of the application that we are working with. We can gain this information from the application itself, however being able to directly see the information in the database can help us as well. This ability would be especially useful because currently we can only use CLI tools for which we have to log into the pods first, which could be considered an obstacle in some cases. Furthermore, terminal output of the CLI tools is rather difficult to navigate, and work with. Therefore, having a more user friendly UI would be welcome.

To summarize, bugs are a common occurrence during software development, and because of their frequency, being able to debug them faster would improve the maintainability of the platform and its applications. Especially useful would be a way to monitor resources, and various metrics, while also being able to search through logs faster, and find relevant information. Additionally, improving the process of viewing the state of our database would help us better understand the current situation as well.

2.3 Misconfigurations

Configuration is supposed to specify how an application should behave. There are different situations where applications should behave differently, whether that is saving files in specific paths, or sending email to various email addresses. Misconfiguration then means that the application is using either configuration that instructs it to behave not as the users intended, or the configuration defines a behavior that the application cannot achieve, given the current situation.

One of the first cases of misconfiguration we have met with were the default configuration settings. The settings are usually chosen to be useful for most people, so that they do not have to change them. However, as the complexity of our platform keeps growing, we have to adjust a few settings. Unfortunately, this requires adjustment of other settings as well, which we did not know at the time. In the end, we are eventually met with errors due to the misconfiguration. For example, one of the possibly most common default settings that we had to change over time were file paths, maximum limits for file sizes, and reference names. As a simple example, the default *etcd*[5] database size is by default limited, and with time we have hit this limit, and we had to adjust it[1].

Kubernetes has many parts that require configuration. Whether it is the applications themselves, or defining how the internal network and routing should behave. With time however, the complexity of the Kubernetes cluster grows, and with it, the configuration keeps changing. Unless we stop improving the cluster, the configuration will keep changing to meet the new requirements.

Of course in many cases automation can help decrease the frequency of misconfiguration. We are able to use environment variables instead of hard coded values for our applications. Thus, we simplify the complexity of our applications. We will no longer have to keep adjusting configuration in three different places, we will have to change it only once.

There are however limits to what parts can be simplified in this way, and many cases require manual adjustments by developers. For instance, when defining what domain name our application is hosted on, we are not able to automatically register a domain name, and then forward this information to our application. In a similar way, some projects have constrained budgets, therefore they have to use less powerful VMs. The software is however unable to read this information, and it is up to the developers to translate the current situation into the configuration.

There are also cases where automation is possible, but requires considerable effort to implement. In these situations, automation would be an ideal outcome. However, if we

[5]https://etcd.io/

were able to estimate whether the effort required to implement the automated solution is high, and yet we expect to rarely modify the configuration, we would rather change the configuration manually when needed. Therefore, saving implementation efforts.

There are various reasons for why misconfiguration is a common occurrence. Developers do not always have access to perfect information, there can be human mistakes happening, miscommunication can occur, or we simply forget to adjust a specific setting. Developers can make mistakes especially as the configurations grow larger. We usually cannot afford to read all the configuration options, primarily, if we are making frequent changes. The documentation can point us to the relevant settings, but even there we have to hope that documentation is kept up to date, and not lacking crucial details.

Developers are not the only cause of misconfiguration either. In many cases logical errors in the software can result in misconfiguration as well. There can be cases of file corruption, or unexpected process termination which leaves the configuration inconsistent. Additionally, it is overall difficult to detect misconfiguration[7].

Specifically in our case, we have numerous places that hold some kind of configuration. We have to set hardware requirements for our infrastructure, its geographical location, security, which machine learning models we should use, what domain name we are using, or where to download Docker images. Developers have to remember all of this information, and where it is set, especially if they have to adjust the configuration on a request of other team members who are not experienced in IT.

As we have mentioned, it is possible to decrease the amount of misconfiguration with automation. It is not a panacea though, and configuration will not stop being a problem any time soon. We are bound to encounter problems, and these problems can be quite expensive in later stages of development[43]. Moreover, the sooner we are able to resolve them the more time we will have to improve other parts of our platform.

As we have mentioned, misconfiguration is a rather frequent problem in cloud environments [44, 47, 45, 36]. Theoretically, if we were to never change anything in the platform, we should never have problems with misconfiguration. Therefore, we are able to some degree influence the misconfiguration rate. Still, we plan to further expand our platform, and the bottom line is that external services develop as well, thus over time we will have to eventually change some settings.

The impact of misconfiguration is very varied. It can cause trivial annoyances, but also a complete dysfunction of our platform. Although, at the moment everything is fully functional, and therefore we are able to roll back any new changes if we are having troubles solving misconfiguration, we still believe that it is quite crucial to find ways in which we can help developers debug misconfiguration issues faster.

Specifically, we need to focus on two areas that need help. When an error occurs, we need to know what was the state of the configuration at the time of the error, or at least, we need to know the most recent configuration before the error. Additionally, we need to understand why that configuration caused an error. That way, we can start trying to find a resolution as early as possible. Fortunately, as it often happens, applications print a lot of the information that we need into the logs[6]. Although, it becomes a problem to find the useful information in a heap of other log messages.

We have various areas where we are using configuration, and where it is crucial for the configuration to be correct. When creating infrastructure, we use Terraform to define all resources that we need. Naturally, these resources all have to be correctly configured, as otherwise our hardware might not be capable of hosting our applications. We are not able to improve the debugging speed in this case however. When we are deploying the infrastructure, we are doing so manually. If there are any misconfigurations, we can see the error messages right away, and we rarely change the infrastructure settings. Therefore, we know the current configuration immediately, and we have access to the error messages right away. As such, we are not able to speed up debugging in this case.

In an almost identical fashion, we use Kubespray to successfully deploy Kubernetes onto our infrastructure. Arguably, the logs that are quite large in size, as a successful deployment lasts around 30 minutes, and it is constantly creating new log messages. This deployment is however done manually as well, and any error messages terminate the deployment immediately. Therefore, here we are not able to help much.

There is a case with Kubespray which we can improve however. When using the autoscaling Kubernetes feature described in section 3.7.2, we need to collect the logs into history. Otherwise, we lose all of the information describing why the scaling did not work as intended. The logs are especially large because we are using high verbosity to fully understand what happened, and what commands did not succeed. Moreover, we have to automatically generate the configuration for auto scaling. Meaning we change a lot of configuration, and it has to be correct in order to work. Thus, we print out the configuration every time before and after scaling, so that we can always know the configuration that failed. As such, knowing the log history, and being able to search it would be very helpful.

One of the main cases where improving the speed of debugging in case of misconfiguration would be helpful are our applications. We define most of the configuration using YAML files, so theoretically we always know what configuration was used during an error. There are however multiple places where the configuration can be specified, and each has a different priority. Thus, we cannot always be sure what configuration was used. If the configuration is printed however, we are able to view it in the logs, and know exactly what settings are used. For example, the following information is printed when we initiate our web application:

```
2021-05-31 11:06:19,263 DEBUG Using selector: EpollSelector
2021-05-31 11:06:19,558 INFO HTTP/2 support not enabled (install
    the http2 and tls Twisted extras)
2021-05-31 11:06:19,557 INFO Starting server at tcp:port=443:
    interface=0.0.0.0
2021-05-31 11:06:19,559 INFO Listening on TCP address 0.0.0.0:443
2021-05-31 11:06:19,559 INFO Starting factory <daphne.
    http_protocol.HTTPFactory object at 0x7fde8ae9faf0>
```

This information can be quite valuable to us, but only if we are able to find it. Therefore, it would be beneficial to somehow store these logs for longer periods of time, and provide tools that would allow us to search them. Especially, since our applications can create a lot of redundant logs that complicate the search using some simple tools.

Yet another place where we can improve the configuration of the applications is by browsing Kubernetes resources like Pods or Deployments that define configuration of the application. We are able to view this configuration when using terminal based tools, however a lot of information in their output is generated by Kubernetes. Therefore, we have a lot of information that we are simply not interested in. If we could somehow reduce these large amounts of automatically generated configurations, and streamline the editing of the configuration, we would improve the debugging process considerably.

To summarize, we have cases where it is not viable to improve our debugging capabilities, such as when creating the cluster. We have, however, a lot of cases where could improve the search and storage of logs, in addition to simplifying the modification of our applications. Improvements in these areas would considerably make our debugging efforts faster, as we would create adjustments faster, and especially find the underlying causes of problems.

2.4 Performance and Traffic Overload

Rather common events that cause cloud incidents are unexpected performance spikes. Applications have to run on top of some hardware, whether that is your personal computer, or large data centers. Naturally, performance of the hardware varies widely from very basic CPUs to highly advanced CPUs, and each offers a different processing power. Unfortunately, each hardware has its limits.

When we reach these limits, we are simply not able to process more information than we already are. In ideal cases the requested computations are saved in memory, and over time processed in a queue. However, even memory has its limits, and if we reach those, there are no simple solutions to handling such cases, and ideally we would completely avoid such situations.

The causes of increased performance or load are numerous. It can be caused by natural increase in requests from the user side, accidental misconfiguration, bugs, but they can also be maliciously caused Denial of Service (DoS) attacks which aim to overwhelm your service, and therefore create performance spikes[44, 47, 48, 30]. Overall, the sources are so varied that prevention alone is not enough to avoid such problems.

Moreover, the distributed nature of the Kuberentes platform means that our applications need to communicate between each other, especially using HTTP. Usually, it does not create an unbearable amount of traffic, but it surely is higher than when working with monolithic applications. Additionally, there are certain cases where we have to deal with problems that are specific to distributed problems.

A concrete example in our platform would be the Kubernetes functionality itself. As it is a distributed system, Kubernetes has to communicate with the VMs it is running on. If it is not able to do so, it will assume that the VMs no longer exist, and this can cause a whole myriad of problems. As with any other communications, most of the requests have some timeout period after which they are assumed to be lost. As such, performance spikes can cause these communication requests created by Kubernetes to timeout, as they might take too long to be processed. The result is that Kubernetes will report that it cannot communicate with the VM, and it will return an error as you can see in the logs below:

```
I0123 22:48:25.733770       1 shared_informer.go:230] Caches are
    synced for client-ca::kube-system::extension-apiserver-
    authentication::client-ca-file
I0123 22:48:42.132747       1 leaderelection.go:252] successfully
    acquired lease kube-system/kube-scheduler
E0123 23:09:28.270983       1 leaderelection.go:320] error
    retrieving resource lock kube-system/kube-scheduler: Get https
    ://10.0.1.6:6443/api/v1/namespaces/kube-system/endpoints/kube-
    scheduler?timeout=10s: net/http: request canceled (Client.
    Timeout exceeded while awaiting headers)
I0123 23:09:28.271045       1 leaderelection.go:277] failed to
    renew lease kube-system/kube-scheduler: timed out waiting for
    the condition
F0123 23:09:28.271059       1 server.go:244] leaderelection lost
```

Needless to say, we want to avoid any errors in our platform. Performance issues require attention of developers, and if such incidents result in further chaos, the developers have to spend even more time on trying to make the platform operational again. Especially, if errors in one application can cause data loss or inconsistent states that are very hard to deal with. A quick example would be when our web application had a bug which created a lot of connections to our database. In the end, we fixed the bug, but our database was inaccessible to other applications as a result of the bug, and we had to recreate the database, which resulted in data loss.

With this in mind, if we were able to provide an automated handling of performance spikes, we would not only free up developers' time, but also increase the maintainability of our platform. The reason being that if we will handle the issues before they become a problem, there is no need for developers to bother with fixing.

To further describe the current situation of our platform, we are able to handle usual traffic, and even handle increased traffic to some degree. Unfortunately, any unexpected increases in traffic would likely overwhelm our services. The main reason for our lack of capability is that we have never explored this area in detail yet. We know that our applications are able to handle quite a few users per second, but so far we were mostly focused on creating new features rather than performance.

So again, we currently have the infrastructure to handle usual traffic, and even some slowly increasing traffic, but we are not ready for sudden traffic spikes. Specifically we have three main areas where we should consider our performance capabilities. Those are the regular web traffic, the video upload, and the machine learning model processing.

In terms of numbers, our web application receives the most requests on our platform, as it faces the Internet, and it holds most of the used functionality by the users. The requests are however simple to process, which means that we are able to process them much faster, and thus, we can serve quite a few users per second.

As we have mentioned, the problem exists because our hardware is physically incapable of processing all the requests, and to avoid running out of memory problems which would outright terminate the application, we have to ignore, timeout, or deny the requests.

Therefore, if we are not able to adjust to the increasing traffic, especially unexpected spikes in traffic numbers, we can easily be overwhelmed. Unfortunately, not being able to serve our website to the users is a major issue, as they are not able access the rest of the functionality.

Naturally this is very problematic, as currently, we do not have any ability to automatically increase the number of applications. Therefore, as soon as we experience a traffic spike that our developers will not notice, the web application will get overwhelmed and crash, even though we have hardware resources available. Therefore, automatically increasing the number of concurrently running applications would help as well.

Furthermore, we could create a caching service, which would be optimized to quickly serve the most requested websites. Thus, it would help our web application serve requests to the users. However, it might be difficult to implement, and configure properly. Moreover, we would be able to lower the performance requirements for the web application, but we would have to eventually increase our hardware performance anyway.

As such, a simpler solution should be to directly increase the hardware resources that our platform can then use. The Kubernetes should be able to provide this functionality without any complications, as they list it as one of their main features[23]. Therefore, automated increase of hardware resources based on the incoming traffic, should be able to increase our maintainability considerably, as we would be theoretically capable of managing any amount of requests.

We allow the beekeepers to upload their videos in two ways. They can either automatically upload the videos with their cameras through our FTP server, or they can use our website to upload them manually. For both cases, a nonfunctional upload of videos would result in major problems because the analysis of videos is one of our core functionalities. Users will still be capable of using the platform, but our main feature will be inaccessible.

Fortunately, if one of the two options for uploading fails, there is always the second option available. Additionally, we do not expect to have problems with our FTP server being overwhelmed. We allow access to our FTP server only for registered users, therefore, we would be able to adjust the available performance for our FTP server based on how many users we would have.

Moreover, the web upload communicates directly with the Azure servers where we store the videos. As such, as long as the users are able to access our website, uploading the videos would be functional. Of course the assumption is that Azure will be able to handle traffic from our users, but considering that they are one of the main cloud providers, we expect them to have no problems. Therefore, improving the web application performance capabilities improves our web upload as well.

In terms of FTP upload, we do not expect to have problems there either. Our main performance bottleneck is the network throughput because it is the only resource we consume considerably. Our FTP server simply forwards the traffic to our Azure storage, so CPU and memory usage is minimal. Fortunately, the VMs that we are currently using support 200MB/s traffic throughput, and our current traffic is around 3MB/s to 6MB/s [31]. Thus, the easiest solution, especially considering that it solves other problems as well, is creating new VMs when needed.

The last and somewhat special case where we need to consider performance is processing the videos through our machine learning models. To process the videos quickly enough we have to use GPUs, and their configuration is not as simple.

Fortunately for us, we are capable of managing the performance requirements as we desire. Therefore, we do not expect to experience any performance spikes that would render our video processing application dysfunctional. We want to of course maximize the performance, but currently we have a queue of videos that need processing. This queue is being slowly depleted, and if we want better performance, we can add new hardware resources as necessary. Thus, apart from automating the process of adding new resources, we were not able to find any long term maintenance improvements.

To conclude, we have a few sections of our platform that need to stay operations as they represent our core functionality. Many of them however, do not have any clear and easy to implement solutions that would be worth pursuing in our case, apart from simply increasing our hardware capacity. Most of the performance that matters for us, we are able to manage, therefore, we are not necessarily forced to adopt complex automatization solutions, and automating the process of adding new hardware resources should be the best solution we can implement.

Fortunately, Kuberentes is able to help exactly with these cases, and it is also a reason why we created our platform on top of it. It is not easy to write proper concurrent applications, and Kubernetes helps us by creating clones of our applications, which are then working in parallel. Moreover, Kuberentes can help us automate adding new hardware resources where our applications can be executed.

Additionally, if we are able somehow better understand the current performance of our platform, we would be able to much better predict the future performance requirements and growth. A solution that would allow us to see what is the current network throughput, CPU usage, or application resource usage would help us immensely in better understanding the performance problems we might be having.

2.5 Failure of External Services

Many cloud platforms or applications consume other services and functions through APIs. These are provided either internally by other development teams, or externally by different organizations. In many cases the cloud platforms are built on top of these services. Meaning that they represent crucial functionality of a given platform, and without them, the platform is simply not operational.

These external services are in many cases themselves hosted on cloud infrastructure, and are just as vulnerable to problems as we are. Therefore, temporary outages are still possible no matter how important the service might be. One could argue that the cloud environment is supposed to help with exactly this issue. There should be no single point of failure in your applications, there should be no single API endpoint that you have to rely on for your functioning. Theoretically, this is actually true as many crucial APIs are hosted on multiple virtual machines, and in different geographical zones. However, special circumstances can still cripple all of these distributed services at once[41, 2].

When we are talking about external service, this can include Internet Service Providers (ISP), storage infrastructure, network infrastructure, source code hosting, DNS resolution service, user authentication and authorization, load balancers, or URL redirection services among many others[45].

Our platform depends on many services as well. This includes somewhat crucial services such as, GitHub, GitLab CI/CD, or DockerHub. Fortunately, we are not completely dependent on their functionality, meaning, if we do not modify or in any way change our applications, we do not need these services. At the time of writing however, we are using specifically three services that can be considered crucial for our operations. Those are any infrastructure APIs provided by Azure, DNS resolution by Hostek[6], and provision of SSL certificates by Let's Encrypt[7].

Concerning the less crucial services, they do not have a wide ranging impact on the functionality of our platform. Specifically, the services that we have listed are used for creation and modification of our cloud environment. Therefore, even if these services were down for multiple weeks, we are still able to postpone the updates of our platform until the services are again fully functional.

As such, it is not crucial to provide any extra information for the developers in terms of debugging such incidents. The outages will result in delayed updates from our side, perhaps we might even manually disable the automatic updates so that the developers are not swarmed with repeated alerts that updates have failed. However, outside of this mitigation, the developers simply have to wait for the services to be operational again. In the end, there is no change needed, as the updates will resume and succeed, after the external service is fixed.

Additionally, although it might be easy to detect unreachable APIs by a simple HTTP request that would fail or succeed, the services can still experience problems where the output is inconsistent, malformed or incomplete. In these cases detecting such issues would be problematic, as the output is often changing, and knowing what should be the response is difficult. For example, we can detect that the GitLab is providing access to our repository, however we are not able to easily determine whether the response is not a corrupted source code. Therefore, the effort required to debug these issues outweighs the benefits, and we will not be focusing on them. However, the more crucial services used in our platform deserve a further analysis.

Starting with the SSL certificates provider Let's Encrypt. Currently, the main website[8] sets the Strict transport security policy header[9]. This HTTP header requests the user's web browser to block access to our website, unless the browser receives valid SSL certificates.

This is supposed to protect the users in case our certificates become invalid for any reason. For example, there could be a security breach of our servers, and the attackers might steal the encryption keys that allow them to decrypt user's connections. If we

[6]https://hostek.com/
[7]https://letsencrypt.org/
[8]https://www.beelvingsensor.eu
[9]https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Strict-Transport-Security

were to detect this breach, we should then contact the Validation Authority[10], and they would mark the certificates as invalid.

However, in case the certificates become invalid in other ways, such as the end of validity of a certificate or domain name change, this makes our website unavailable for the users, although there are no threats present. This means that if we are not able to renew our certificates in time, our website would become unavailable. The risk of this happening is rather low, as the validity duration of the certificates is 3 months, and they are automatically renewed every 2 months. As such, the Let's Encrypt services would have to be unavailable for a month, before our website would become unavailable. In this case we believe that a month is enough time for the Let's Encrypt services to recover, or at the very least for us to detect, and resolve this issue without bigger problems.

A more crucial service is our DNS provider Hostek. Without a working DNS resolution, the users will not be able to translate our domain name into an IP address. Moreover, regular users would not be allowed to access our website even if they knew our IP address, as our Django web server has a host filter enabled. The host filter is responsible for accepting only HTTP requests that specify domain name in their headers. If the header is not defined or if it uses a different domain name than expected, the HTTP requests are ignored. As such, the users would have to create their own HTTP requests manually, which is beyond the knowledge of every day users. This behavior is supposed to prevent cross site request forgery among other attacks[5]. Overall, the probability of a DNS outage is rather low as DNS servers are a very mature technology, but it still can happen[33], and it would make most of our services dysfunctional.

Fortunately, it is rather simple to detect the unavailability of a DNS resolution. In general, it is possible to execute one of the many DNS resolution tools available on Linux and Windows, such as *nslookup*[11] or *dig*[12] in a loop, and as long as they succeed, the DNS works. In our case however we can support the developers in an even better way. The Hostek itself has a status page[13], and there they allow anyone to subscribe to status updates. Therefore, as soon as there are any issues that could result in a suboptimal performance of the DNS, the developers will be notified. In terms of debugging a DNS outage by the developers, running any of the previously mentioned tools would clearly state that the DNS server could not be reached.

Finally, the most important external service we are using is Azure. The Beelivingsensor project received a research grant from Microsoft, in the form of Azure credits, and thus the whole cloud platform is hosted solely on Azure infrastructure. This means that any problem that Azure might have with any of the services we consume, such as storage, networks, Kubernetes APIs, we will experience it. The impact of any of the services being unavailable would simply result in our platform not functioning. At the time of writing, we are using three types of services from Azure, storage, compute resources, and networking. Issues with any of these services would mean that our platform would be unusable, as all of them are crucial for our functionality.

[10]https://datatracker.ietf.org/doc/html/rfc5280
[11]https://linux.die.net/man/1/nslookup
[12]https://linux.die.net/man/1/dig
[13]https://hostek.com/serverstatus.html

After some research, the frequency of outages in any Azure service is rather common. Specifically, there is an issue with at least one Azure service once a month[12]. However, considering the large number of data centers and services that Azure provides, this is a rather small number overall[11]. We should not however underestimate these outages. Even from our experience, we have already had to deal with at least two critical outages in the last year, which resulted in an unusable platform.

In terms of detection, spotting an outage would be fairly difficult considering the extensive number of Azure services, and their data centers. However, Azure does provide all of their users with a *health* service, free of charge[22]. This service is capable of notifying users of any planned or unexpected events that might disrupt the services they are using. Therefore, detecting such events becomes simpler, and especially flexible, as the users can set up email, SMS, or other notifications in case of incidents. The more difficult part is debugging these issues. During an outage it is not always clear whether all of the services or only a subset of them are inaccessible. Additionally, there is a possibility of stale or inconsistent data between the services. As an example, we had an issue with Azure disks and Kubernetes persistent volumes. The pods were reporting errors as they were not able to mount specified persistent volumes, however the persistent volumes were reporting a functioning state. Only after a closer inspection of the logs for the persistent volumes, we found out that their health probes were indeed unresponsive. The issue was that the timeout durations of those probes did not run out yet, and their states did not report an unresponsive back end.

Moreover, the information that a provider's API is unavailable is visible only through time out logs in the application, or the response message of the Azure API, and therefore error codes of an application. An example log can be seen below:

```
{"level":"warn","time":"2021-02-25T19:31:55.814","sender":"FTP","
    connection_id":"FTP_0_19","message":"Unexpected error for
    transfer, path: \"prefix/qqqqeo/2021/02/25/test
    rlc511_01_20210225202130.mp4\", error: \"read tcp
    10.233.64.32:32002->10.233.64.1:12223: read: connection reset
    by peer\" bytes sent: 0, bytes received: 89884202 transfer
    running since 618548 ms"}
{"level":"debug","time":"2021-02-25T19:31:55.815","sender":"Azure
    Blob container \"ftpcontainer\"","connection_id":"FTP_0_19","
    message":"multipart upload error: -> github.com/Azure/azure-
    pipeline-go/pipeline.NewError, /go/pkg/mod/github.com/!azure/
    azure-pipeline-go@v0.2.3/pipeline/error.go:157\nHTTP request
    failed\n\nPut \"https://beelivingsensor.blob.core.windows.net/
    ftpcontainer/prefix/qqqqeo/2021/02/25/test%20
    rlc511_01_20210225202130.mp4?blockid=FgAAAAAAAA%3D&comp=block&
    timeout=240\": context canceled\n"}
{"level":"debug","time":"2021-02-25T19:31:55.824","sender":"Azure
    Blob container \"ftpcontainer\"","connection_id":"FTP_0_19","
    message":"upload completed, path: \"prefix/qqqqeo/2021/02/25/
    test rlc511_01_20210225202130.mp4\", readed bytes: 89884202,
    err: -> github.com/Azure/azure-pipeline-go/pipeline.NewError, /
go/pkg/mod/github.com/!azure/azure-pipeline-go@v0.2.3/pipeline/
error.go:157\nHTTP request failed\n\nPut \"https://
beelivingsensor.blob.core.windows.net/ftpcontainer/prefix/
qqqqeo/2021/02/25/test%20rlc511_01_20210225202130.mp4?blockid=
FgAAAAAAAAA%3D&comp=block&timeout=240\": context canceled\n"}
```

As can be seen, the upload of a file through FTP into Azure storage did not succeed. Moreover, without an extra debugging flag set by the developers, the only hint that an API is misconfigured, and does not accept traffic, would be the first *warning* level message that is rather unspecific. Since this error has *warning* level importance the application will not report an error, although the crucial functionality is unavailable.

Unfortunately, the logs have their own set of disadvantages. The main one, as you can see in the example logs, is that in some cases the log level of an error is only *warn*, short for *warning*. This means that our application continues to run, but it does not actually warn the developers about any issues. As such, the developers will not be notified of any errors, unless they decide to browse the logs. As such, there is no fast and unified way to debug these incidents, outside of the Azure *health* service notifications, and possibly viewing the logs.

3

Maintainability improvements

This chapter answers the RQs that we have defined in the section 1.2. Our initial goal was to investigate various ways of improving our platform, and then implement at least one of them for each area which we could improve. With time however, we have realized that many of the areas overlap. Thus, implementing one tool would help in multiple cases, and we have decided to simply identify the most beneficial tools, and implement as many as we could. The implemented solutions are described in this chapter, along with answers to our RQs.

To answer the overall RQ 1., each section of this chapter describes ways in which we have decreased our maintenance efforts of the Beelivingsensor platform in multiple areas. Specifically for RQ 1.1., after analyzing the various common problems that we can encounter when working with cloud platforms in chapter 2, we have focused on four approaches that would help us find their root causes faster. Those are a metrics collector in section 3.1, centralized log collector in section 3.2, Kubernetes Dashboard in section 3.3, and health monitor of external services in section 3.4.

For the RQ 1.2. we have searched for a way to detect abnormal events, and then notify the developers as soon as possible, in case they happen. In section 3.5, we have provided an overview of alerts for developers, which decrease the response time for the developers in case of unusual patterns. In the end, the implementation allowed us to considerably decrease the time before the developers found out about an error, and now they are immediately notified about any unusual events.

Solution for RQ 1.3. is provided in section 3.6. There, we have described how we are able to recover from application failures, and how we stay functional even in such cases. Apart from the *self-healing* concepts that we describe there, we believe the section 3.7 on automated scaling also helps answer this question.

In the last subquestion 1.4., we have looked at frequent situations that our developers have to deal with, and our goal was to find ways to automate them, thus saving time and effort for the developers. Overall, we have identified that our developers will have to spend a lot of time on adjusting the size of our computing resources, and since it is also a solution for one of the common problems that we have identified, we have described automated scaling of resources in section 3.7.

Moreover, we have noticed that the process of deploying new versions of our applications could be automated, so as to improve the maintainability of the platform. Therefore, in section 3.8, we have described how we have solved this issue, and how it helped the developers. We have found other ways of improving the maintainability as well, but due to time constraints we were not able to implement them, although they are further described in the future work section 5.3.

Answers to the RQ 2. are included in each section of this chapter individually. There, we described how each solution is beneficial for us, why we have implemented the specific tools, the effort that was required to implement them, and any notable alternatives that we had considered.

In a similar fashion to RQ 2., answer to the RQ 3. is as well included in the individual sections of this chapter. After describing the solutions for each of the areas, we always describe the exact benefits we have observed after implementing a specific solution.

As the chapter 2 explains in more detail, we have obtained the solutions to our questions mostly by exploring various blogs, conference talks, and project repositories like GitHub and GitLab.

3.1 Prometheus and Grafana

In many Common Issues sections we have described how monitoring of our resources would be helpful. We outlined a few metrics, but there are many sources of information that we can consider for monitoring. There are the very common metrics such as CPU usage, memory usage or disk usage, and they are very crucial, but there are others.

Specifically, we have talked about how networks can be monitored, and how watching for unusual patterns using different metrics can help us spot problems. These metrics can include the number of requests that we receive, the size of those requests, or how many 404 responses our Kubernetes APIs return. Moreover, there is basically not even a limit to what metrics we could be measuring. The ones we have outlined so far are only those which are provided in the default configuration, and even the default configuration provides hundreds of sources[10, 13]. Even better, we can create our own metrics to monitor as well. It is rather complex, but as long as we are able to expose a simple HTTP endpoint that uses a certain text format, we can then monitor metrics, such as number of images that are processed, the length of videos that are processed or size of our machine learning models.

The possibilities that we have described above are enabled by the Prometheus[1]. It is not easy to describe what Prometheus is because it is more of a set of tools that users can use based on their needs. Currently, we are its users, and as described in the sections of chapter Common Issues, we need to monitor certain metrics in order to help us debug our applications. However, as useful as the Prometheus might be alone, there is also Grafana[2], which is the second tool described in this section.

Grafana, just like Prometheus, is a collection of many features that users might use as needed. However, we can describe it in simplified terms, as an interactive graph or dashboard creation tool and browser. In other words, Grafana allows us to create, and

[1]https://prometheus.io/
[2]https://grafana.com/

browse many types of graphs and dashboards that are periodically updated with data from Prometheus.

In a similar manner to how our log collection tool works, the Prometheus is responsible for collecting and storing information, while Grafana acts as a visualization of the collected information. Together, the two applications allow the developers to collect important information, and visualize it over time, in order to notice patterns, which help the developers debug their applications.

As mentioned, the Prometheus and Grafana (PG stack) combination is supposed to help developers with debugging their application, and the platform. This is achieved by monitoring multiple metrics, and plotting them over time, so that developers can notice uncommon patterns, which lead us closer to uncovering issues with our applications. Since this section covers the PG stack from the perspective of how we can help developers debug common problems in Kubernetes, we are going to focus on two of the main use cases. This does not mean there are not more benefits from including these two tools into our platform. On the contrary, there are many advantages, and we will be describing how the PG stack helps us, but only in the later sections.

The two use cases that we believe will bring benefits to debugging are monitoring of the network usage, and monitoring of the hardware resources usage. First, we believe that by being able to graph network related metrics, such as bandwidth, number of packets, number of HTTP responses, or packet retransmission rate the developers will be able to pinpoint which applications are causing issues.

Additionally, it is not only the applications that might cause problems, but the underlying infrastructure could cause issues as well. This does not have to be clear from the application logs when observing them. The issue might lie within the applications that are trying to communicate, or with the infrastructure in case it is damaged or misconfigured. In the logs, both causes will be indicated similarly, for example, by an increased number of logs indicating failed connections.

However, with the monitoring tools, we will be able to observe patterns in the networking statistics. In the case of misconfigured infrastructure, we would be able to see that all of the applications hosted on the problematic host are actually having problems, and not just a single application. Conversely, we would be able to observe that a single pod has higher rates of sending requests than the rest. This would easily indicate that only a single application has problems, whatever the reason might be.

The ability to monitor the network resources is crucial for improving the debugging efforts of the developers. Which is especially true, if we consider that monitoring can help with all of the common issues mentioned in section 2, those being performance, application bugs, misconfigurations, overall network problems, and failures of external services.

The second use case for PG stack would be monitoring of the hardware resource usage. In a similar fashion as with network usage, the developers are capable of watching the metrics such as CPU or memory, and they are then able to deduct whether specific issues are affecting a single application instance, or all of them. Thus, they are able to tell whether the issue is only present in some pods, and exactly which of them needs to be debugged, or in all of them.

Moreover, there could be issues with performance. These issues are particularly difficult to detect without monitoring tools like Prometheus and Grafana. The problem being that the applications work without apparent problems, until it is too late and the whole VM, which hosts the application, becomes unresponsive. Theoretically it is possible to use resource monitoring utilities like *top*[3], but this requires a connection to the VM, and even in those cases we would not be able to find which pods have high CPU utilization. Which is also exactly the reason why having a PG stack is crucial, as it allows us to detect, and understand the issues with our applications.

PG stack is very helpful with monitoring, but it was not the first tool we have considered. There are quite a few alternatives, although many of them are proprietary, and paid solutions, while the rest can only partially replace the Prometheus or Grafana in terms of functionality and coverage. In other words, it is possible to substitute the Prometheus, but it would require us to implement a standalone software for metrics collection, and storage, while Prometheus does both, and covers applications and the platform. As for the Grafana, there are possible alternatives, but the community around the PG stack has created already existing dashboards, which we would have to manually create when using the alternatives.

However, the first solution we have analyzed was Netdata[4]. Netdata is able to provide all three components of the PG stack, which is collection, storage, and visualization. It can easily compete with PG stack, except for three reasons, which lead us to adopting the PG stack.

First reason being that the visualization part of the Netdata is hosted by the company which develops it. This means that we are required to create additional accounts so that we can access the visualization, while also being dependent on the availability of their service. While compared to the PG stack, all of the components are fully hosted by us, and we do not require creation of new accounts.

Second reason being more detrimental, which is the difficulty of automation. As simple as the implementation of the Netdata might be, the fact that the visualization is hosted by Netdata company means we have to manually connect the visualization and collection parts of that solution. This has to be done every time the monitoring components are redeployed, which is always the case when we are recreating our Kubernetes cluster. However, with the PG stack, there is no need for this coupling, as the solution we are using can automatically assume that there is an available Prometheus instance present at some URL. Although it could be considered a minor issue, especially since the monitoring components are rarely redeployed, we believe that the automation is too crucial for ensuring that our platform will be maintainable in the future. The less things that a developer has to worry about, the better, and requiring manual setup of the monitoring with Netdata, would be simply a risky situation which we do not need.

Third, while analyzing the monitoring options, we have noticed that the Prometheus includes also the ability to alert developers on certain events. With that in mind, we have looked more into the RQ 1.2., and we have realized that this would suit us very well.

[3]https://linux.die.net/man/1/top
[4]https://www.netdata.cloud/

We have done some further research, and we have concluded that since the effort spent on implementing the PG stack was so low, and we have an opportunity to answer yet another research question, the PG stack was more beneficial than the Netdata option.

There was one more free and open source alternative called Zabbix[5]. At first it seemed convenient, and capable or competing with PG stack. However, Zabbix is not necessarily easy to maintain and deploy. Compared to PG stack, it requires us to deploy and maintain an external database, while Prometheus automatically uses its own builtin database. Additionally, Zabbix is meant to be run as a Linux OS service, which would require us to install it on every VM that we create. This only increases our maintenance efforts, and even worse, compared to PG stack, it requires a lot of effort in order to implement and configure.

In terms of implementation efforts, as we have already mentioned, the effort required to implement a PG stack was very minimal. We have used Helm to deploy various components of the PG stack. There was no configuration necessary in order to start collecting, storing, and visualizing the important metrics. After the deployment, we only had to add an entry to our proxy server and certificate server in order to use proper HTTP routing to the Grafana dashboards.

After the implementation, the expected benefits, combined with how easy the set up was, were very quickly put to good use. During our first deployment, when we were still not sure if everything was working as intended, we noticed unusual patterns in certain metrics. At first we assumed it was simply the case of misconfiguration of the PG stack deployment. However, with time we have noticed that these patterns kept repeating. We were also aware of another unusual error pattern in our Kubernetes services that seemed unhealthy, but we never paid too much attention to it. The pattern was that some of our pods were reporting crashes. These crashes were seemingly random, and the logs of those pods only suggested that there were some communication errors with our infrastructure. Since our other applications were working as expected, we have simply ignored the issues.

With the rollout of PG stack though, we have noticed that these two events, the failing pods and the unusual patterns in certain metrics, were somehow related. The three main metrics we were observing were CPU utilization, disk utilization, and *load average*[6]. Somewhat counterintuitively, we observed that during the crashes of the pods we were monitoring, the load average metric spiked, but the CPU utilization was around 20% per core, and disk utilization never reached levels higher than 10MB/s. Considering that our personal computers, which used hard disks, were able to reach 80MB/s disk utilization, we thought this was low, and not a concern. We were perplexed as to why the hardware resource usage was low, but there were many processes running or waiting for I/O, however we were not worried about it too much.

With time unfortunately, this became a major problem. As we have started deploying more, and more applications, our VMs stopped responding. We have quickly looked at

[5]https://www.zabbix.com/

[6]Load average represents the number of processes in the running or waiting for I/O state over some period of time.

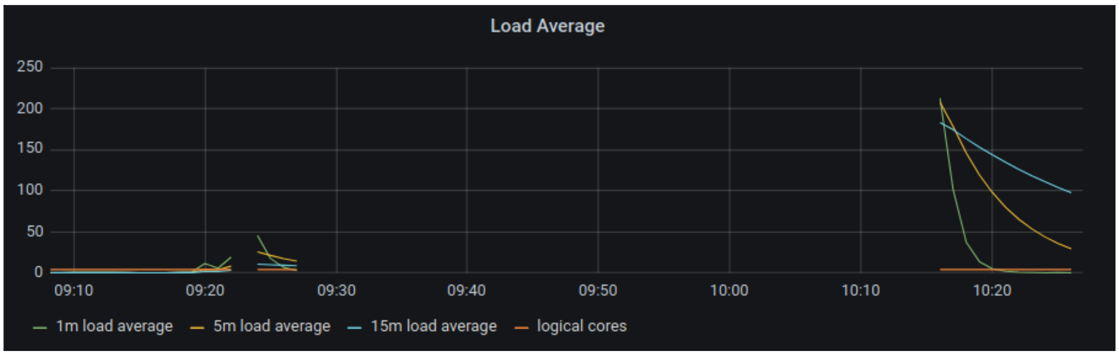


Figure 3.1: Grafana graph showing load average of a VM. You can see a time span when the Prometheus was not able to collect metrics because the VM was overloaded.

the metrics, and yet again we have noticed the correlating patterns. This time however, the metrics spiked so high that the virtual machines were simply not responding, as you can see in Figure 3.1. We have tried using different CLI tools, in order to further debug the situation, but everything seemed in order. The only clue we had were the graphs in Grafana.

With further debugging, we have realized that these patterns were triggered by Docker. When we ordered Kubernetes to deploy a new application, it requested Docker to execute a new container. However, if the Docker did not have the images necessary to start a new container, it had to download them. During downloads, the images were compressed, as many of them were gigabytes in size. Additionally, the Docker parallelized the downloads, and compression in order to speed up the process. This meant that our disk utilization was at its maximum, while the Docker started creating new processes for decompression of the downloaded images. These processes were then waiting for the disk access, and multiplying, while the Docker was still trying to download the images as quickly as possible. The result was a very high load average, and completely unresponsive VMs.

In the end, we have identified that the root cause was a very slow disk. As we have mentioned, our disks never went above 10MB/s. At first we thought there were simply no I/O operations. With benchmarking though, we have uncovered that our disks were strongly underperforming. Azure support was never able to provide sufficient answers as to why this was happening, and we were forced to start using large disks, which did not have this issue.

This issue was very critical, as our applications were unable to deploy. We have tried debugging the issues with CLI tools, but we were unable to notice these patterns. We believe that without Prometheus we would have spent even more time trying to debug this problem. Without the very practical visualization of graphing different metrics over time, we would not have been able to spot that the load average, and unresponsive nodes, were related. Especially, if we were not able to obtain these metrics through CLI tools. Afterall, the VMs were outright inaccessible.

So far we have not encountered any issues which were concerned with the network only, but we used the PG stack again when our web applications stopped working. In this case some of our requests to our *model* service, which handled machine learning models, were timing out. Logs inside the model application did not show any problems. On the other hand, as soon as we opened our PG dashboards, we could clearly see that our memory utilization was close to 100%. We could also easily correlate the time when the memory usage started to become a problem, and the moment when we started using the *model* service. We finally confirmed this with a detailed memory usage of the pod that contained our *model* service. The further steps were to run a profiler on this service. This situation shows how Grafana dashboards considerably increased the speed of debugging this issue.

To summarize, we were able to harness the benefits of our PG stack very quickly. It allowed us to view metrics history and up to date performance. This meant we were able to observe unusual patterns, and possibly correlate the patterns with other events. Moreover, we were now able to easily monitor all VMs in our cluster, without the need of having direct connections with each of them. It was also now much easier to debug performance problems, which we were unable to spot by reading logs, and the Prometheus enabled us monitoring of metrics that were previously unavailable, such as resources used per application.

Although the PG stack is a very valuable tool, there are still a few shortcomings that we have encountered. First, some of the preconfigured dashboards do not show data. This can be caused by a myriad of problems. For example, the scraping formats that Prometheus requires might be incorrect, or the Prometheus might not have permissions properly set for scraping the targets, or the targets are simply not running. Understanding why the metrics are not available requires understanding how the Grafana dashboards are compiled, which is currently not our top priority. Therefore, we are not using the full potential of the PG stack, and the effort to solve it is not trivial.

Second, the reuse of customized dashboards is not trivial either. It is possible to build customized dashboards using a built in Grafana editor, and then extract them. The problem is in automatically including the previously created dashboards in new PG stack deployments. Because of how closely integrated the Grafana and Prometheus are, we would need to adjust the process of compiling these dashboards, which means more maintenance effort from our side in the future.

Lastly, if we were to expose our custom metrics, such as the number of images processed through our *model* service, we would also have to build customized Grafana dashboards. Which is a problem by itself, but we would also have to additionally adjust the configuration of Prometheus. Afterall, the Prometheus has to know where the data is exposed. Therefore, in order to use custom metrics, we would have to correctly expose these metrics, then adjust the Prometheus configuration, and finally prepare and compile Grafana dashboards.

3.2 Centralized Logging

Applications create a lot of logs over time. Even our rather basic applications create around 25MB of logs every two days, and the more complex ones around 4GB. Therefore, having multiple applications, and their replicas running at the same time can create a lot of logs. Especially, if the applications run many months without stopping.

In just a few days our logs are rendered too large to be manually searched. Every day our applications are accepting traffic from web crawlers or automated botnets that are trying to find vulnerabilities in our website. Every request is logged for purposes of debugging, and currently we are not able to tell apart spam, and actual requests to our website. This means that our logs are quickly filled with redundant information, and therefore much harder to search through. The problem is only increased when you consider that many of our applications have a few parallel instances running along. Additionally, we are running other applications as well, and each creates their own logs.

At the beginning of our research, we thought that our platform is small enough for us to manage searching through logs using just commands that print the logs onto a command line. With further research however, it became clear that we need something in order to help us search through the logs, and especially through multiple services at once. For example, as soon as an application has been running for more than two days, printing of the logs is practically unusable, as a large parts of them are filled with redundant information as you can see below:

```
10.233.64.0:46030 - - [11/Jan/2021:16:22:04] "GET /NASApp/nessus/"
    400 143
10.233.64.0:46146 - - [11/Jan/2021:16:22:04] "GET /" 400 143
10.233.64.0:46146 - - [11/Jan/2021:16:22:04] "GET /" 400 143
10.233.64.0:45930 - - [11/Jan/2021:16:22:04] "GET /" 400 143
10.233.64.0:46014 - - [11/Jan/2021:16:22:04] "GET /" 400 143
10.233.64.0:46146 - - [11/Jan/2021:16:22:04] "GET /query.idq?
   CiTemplate=../../../../../winnt/win.ini" 400 143
10.233.64.0:45930 - - [11/Jan/2021:16:22:04] "GET /" 400 143
10.233.64.0:46014 - - [11/Jan/2021:16:22:04] "GET /" 400 143
10.233.64.0:46044 - - [11/Jan/2021:16:22:04] "GET /WebID/
   IISWebAgentIF.dll?postdata="><script>foo</script>" 400 143
10.233.64.0:46146 - - [11/Jan/2021:16:22:04] "GET /query.idq?
   CiTemplate=../../../../../winnt/win.ini%20%20%20%
20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%
20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%
20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%
20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%2
0%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20" 400 143
10.233.64.0:46030 - - [11/Jan/2021:16:22:04] "GET /vsmc.html" 400
   143
10.233.64.0:46014 - - [11/Jan/2021:16:22:04] "GET /builtin/index.
   html" 400 143
10.233.64.0:46044 - - [11/Jan/2021:16:22:04] "GET /WebID/
   IISWebAgentIF.dll?postdata="><script>foo</script>" 400 143
```

```
10.233.64.0:46146 - - [11/Jan/2021:16:22:04] "GET /" 400 143
10.233.64.0:45572 - - [11/Jan/2021:16:22:04] "GET /" 400 143
10.233.64.0:46030 - - [11/Jan/2021:16:22:04] "GET /" 400 143
10.233.64.0:46146 - - [11/Jan/2021:16:22:04] "GET /" 400 143
10.233.64.0:45572 - - [11/Jan/2021:16:22:04] "GET /" 400 143
10.233.64.0:46030 - - [11/Jan/2021:16:22:04] "GET /" 400 143
10.233.64.0:46044 - - [11/Jan/2021:16:22:04] "GET /" 400 143
10.233.64.0:45570 - - [11/Jan/2021:16:22:04] "GET /query.idq?
    CiTemplate=../../../../../winnt/win.ini" 400 143
10.233.64.0:45572 - - [11/Jan/2021:16:22:04] "GET /vsmc.html" 400
    143
10.233.64.0:46044 - - [11/Jan/2021:16:22:04] "GET /" 400 143
10.233.64.0:45570 - - [11/Jan/2021:16:22:04] "GET /query.idq?
    CiTemplate=../../../../../winnt/win.ini%20%20%20%2
0%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20
%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20
%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20
%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20
%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20" 400 143
10.233.64.0:46146 - - [11/Jan/2021:16:22:04] "GET /builtin/index.
    html" 400 143
10.233.64.0:46014 - - [11/Jan/2021:16:22:04] "GET /cs/idcplg?
    IdcService=GET_ENVIRONMENT&IsJson=1" 400 143
```

The core problem is that with such large quantities of logs, it becomes quite difficult to search the logs for any information. After all, having only a few lines to scan would not cause any problems if you are able to read through them in a few minutes, but with pages upon pages of logs, it is impossible to do it in a reasonable time.

Further issues arising from a large number of logs are problems with replicating the errors in case there are multiple instances of an application running. The Kubernetes by default load balances requests, and we never know which application receives it. As such we have problems with having to find out which application received our testing calls.

Fortunately there are solutions for managing, searching, and unifying such large quantities of logs. There are three main parts which are the log collection, log storage and a browser for the logs. These three main components create our logging stack, which is composed of Elasticsearch[7] for storage, Fluentd[8] as a log collection tool, and Kibana[9] for browsing the logs. We will be referring to the collection of these three tools as the EFK stack.

The log collection starts with Fluentd collecting logs from the journaling system of an underlying OS. This also means that the Fluentd has to be present on every VM that it collects logs from. The logs are then simply read by the Fluentd, and forwarded to the log storage. One important point though is that Fluentd also enriches the collected logs

[7]https://www.elastic.co/elasticsearch/
[8]https://www.fluentd.org/
[9]https://www.elastic.co/kibana

with extra metadata such as the timestamp, and an identification of which application created the logs. This allows us to later search through the logs using this metadata, and we are then much more flexible in terms of searching through them.

The Elasticsearch is a distributed document storage and a full text search engine. Meaning it is capable of storing large amounts of text data, and then searching through it in an optimized manner. Therefore, after the logs are collected by the Fluentd, the Elasticsearch processes them and their metadata, and stores them for later. Because Elasticsearch is also distributed, it is very suitable for cloud environments, and it automatically solves our worries with data replication, as now we no longer have to worry about a single Elasticsearch failing and losing all of our data.

The Elasticsearch also allows us to quickly search through the text, and this functionality is used by Kibana. As we have mentioned, Kibana acts as a browser for the logs. It does not do the searching, that is part of the Elasticsearch, but it is used for creating logical queries for the logs, and especially for displaying the logs in an easy to read format. Kibana is also capable of creating various dashboards that periodically summarize the logs that we collected. With this functionality, the developers are then capable of creating summaries of the logs that suite their needs. For example, we are able to create a dashboard that can count the number of exceptions that occurred in our application, and this can help us quickly determine whether there are any issues that we should investigate.

To summarize, these three tools are closely integrated together, in order to create a pipeline that automatically collects and stores the logs over long periods of time. Moreover, they allow us to create logical full text queries which help us with browsing large amounts of logs.

EFK stack provides us with multiple benefits that we have already mentioned, but there are more of them. One of the especially useful advantages is that the Elasticsearch stores also logs for pods that were already deleted. Since the pods are basically just clones of a single application that are able to run in parallel with each other, there are situations when some pods simply crash, and Kuberentes deletes them.

This complicates the retrieval of logs for developers when the pods are deleted, as they need to browse them in order to debug a problem from a few days ago. Technically, the logs are still present on the VM, but the developers would have to manually find the compressed logs first, and then somehow read them[8]. Of course this is not optimal, nor is it a quick and easy solution. However, this is exactly the situation where the EFK stack becomes useful, as it keeps the log history, and we are able to search it just like any other logs.

Moreover, as we have already mentioned, any application can be deployed in multiple pods. This somewhat complicates the situation when developers are trying to replicate bugs, as Kubernetes automatically load balances the incoming traffic into all pods, and the developers are then not able to tell which pod received the traffic. With the EFK stack however, it is possible to search all pods relating to one application, and therefore, we no longer have to print logs of every pod in order to find the one we need.

This leads us to another issue that EFK stack helps us with, which is the performance. We mentioned that over two days our applications collect more than 4GB of logs. As you can imagine, our pods can run for many weeks, and even months without any problems, which results in a lot of logs. Therefore, if you are trying to print the logs into a file using the terminal, it can actually take a few minutes to even physically print them into the file.

Additionally, if you consider that many times you are trying to search logs of multiple applications at one, this quickly accumulates into an even larger amount of logs that you need to print, and then search. Many times, when debugging, we also need to search the logs according to a timestamp, and see whether any other services experienced problems as well. Of course, if we are going to search all pods running on Kubernetes this becomes practically impossible to do manually. However, as the Elasticsearch is directly optimized for full text search, it has no problems processing even GB of data. Additionally, since the log metadata contains timestamps, we are able to narrow down the search time window.

During our research we have also managed to find a few alternatives to the EFK stack. The most prominent being the combination of Promtail[10], Loki[11], and Grafana. In the same fashion as the EFK stack, the Promtail functions as a log collector, the Loki as a storage, and Grafana as a log browser. The main functionality is very similar to the EFK stack as well, but there is a difference as to what these three components aim to achieve, especially Loki. The belief is that with very large Kuberntes clusters it becomes difficult to manage logs at scale. Therefore, Loki does not allow indexing of the log's text, only the metadata that Promtail added to the logs. Which is a valid concern, and Loki aims to solve it.

Our situation is a bit different however, and we believe that searching text of the logs is very valuable for debugging purposes. Currently, we also do not have any problems with performance, and there are no plans to have a very large cluster running. Therefore, the Loki's use case does not apply to us, and we have decided to use the EFK stack.

The EFK stack is made of three components. These components are tightly integrated, and supposed to work together, but there is always the possibility of replacing one of the components with a different application. This is also exactly where we found most of the other alternatives to be situated. We have explicitly mentioned the Promtail, Loki, and Grafana stack only because it was sufficiently different from the rest. However, it is always possible to replace the Fluentd in EFK stack with any solution that is able to forward logs to the Elasticsearch. The Fleuntd can be replaced by Promtail, Logstash[12] or Rsyslog[13], and the same goes for the Elasticsearch, and Kibana.

Having so many alternatives results in a very large number of possible combinations, while many of them still contain the same functionality that allows us to search through the logs. Of course there are too many combinations to list, and especially describe, while many of them are still valid replacements for each component. In the end, we have decided for the EFK stack simply because of its effortless implementation, and tight integration, which is described in detail later. To the best of our knowledge, we were not

[10]https://grafana.com/docs/loki/latest/clients/promtail/
[11]https://grafana.com/docs/loki/latest/
[12]https://www.elastic.co/logstash
[13]https://www.rsyslog.com/

able to find any outstanding alternatives that would lead us to reconsider our current setup.

Since we are deploying an EFK stack, as the name suggests, we need to install three main components, Elasticsearch, Fluentd, and Kibana. The First component being the Fleuntd. It is responsible for collecting logs from the journaling system of the underlying OS, and forwarding the logs into the storage component that is Elasticsearch. With Fluentd the installation was straightforward, as the only configuration we had to customize were the authentication details for the Elasticsearch, and where to find the storage. The implementation was simplified even more, as we have installed it using Helm.

Second component was the Elasticsearhc itself. In this case we used Helm for installation as well, however here we had to customize a few settings. The first rather simple options were physical storage for the logs, and account set up. We used Kubernetes to automatically provision hard drives for each Elasticsearch database instance, and we have specified a simple authentication name and password.

The harder part came with securing the communication. Elasticsearch is used as a storage for the logs, however we need something to browse them. This is where Kibana helps us. Unfortunately, Kibana was programmed so that authentication to browse the logs was enabled only in case the communication between the Elasticsearch and Kibana was secured with SSL encryption.

This meant that if we did not use HTTPS certificates for communication between Kibana and Elasticsearch, the Kibana website would be accessible without requiring username and password. Therefore, anyone who would know the URL for Kibana, would be also able to browse logs of any application running in the cluster. This is of course a security risk as the logs might contain sensitive information used for debugging purposes.

At first we tried to somehow force the user authentication for Kibana. Afterall, the communication within the cluster was using HTTP only, and the communication between the user and our cluster was properly secured. However, after a long research we have found out that the authentication is not for using Kibana, but actually for accessing the Elasticsearch cluster. Meaning that when Kibana provided a login screen, we were not authenticating to Kibana, but to Elasticsearch. This explained why the authentication was rather forced, although in our case we needed to authenticate with Kibana only. Therefore, we had to enable the security features on Elasticsearch.

Further complication was that with the security package enabled, the Elasticsearch required SSL certificates for communication. We have tried to manually create one-time certificates with *openssl*[14] CLI, however we were not successful, and the Elasticsearch marked them all as invalid.

Our other option to create valid certificates was using a tool provided inside the Elasticsearch image. We wanted to avoid this however, as using that tool meant that automating the certification creation would become cumbersome. To create them automatically with every deployment of the EFK stack, using this tool, we would have to

[14]

execute the following steps. First, start the Elasticsearch database cluster, then create the certificates in one of the Elasticsearch instances, and then share the certificates with the rest of the instances. Only after each instance had valid certificates, was the whole cluster capable of proper functioning.

In the end, we have used the tool from Elasticsearch to create the certificates, but we have extracted them, and saved them in our repository for later use. So every time we create a new cluster we reuse the old certificates. This should not be a problem in general, but if they become invalid for some reason, they would have to be manually generated again, which theoretically decreases the maintainability of the cluster. Overall, this was the most complex part of the installation.

Last component that needed installation was Kibana. Just like with the two previous components, we have installed it with Helm. The configuration was simple as well. We had to configure an account that was used to connect to the Elasticsearch, and add a new subdomain to our routing configuration which would be used to reach Kibana.

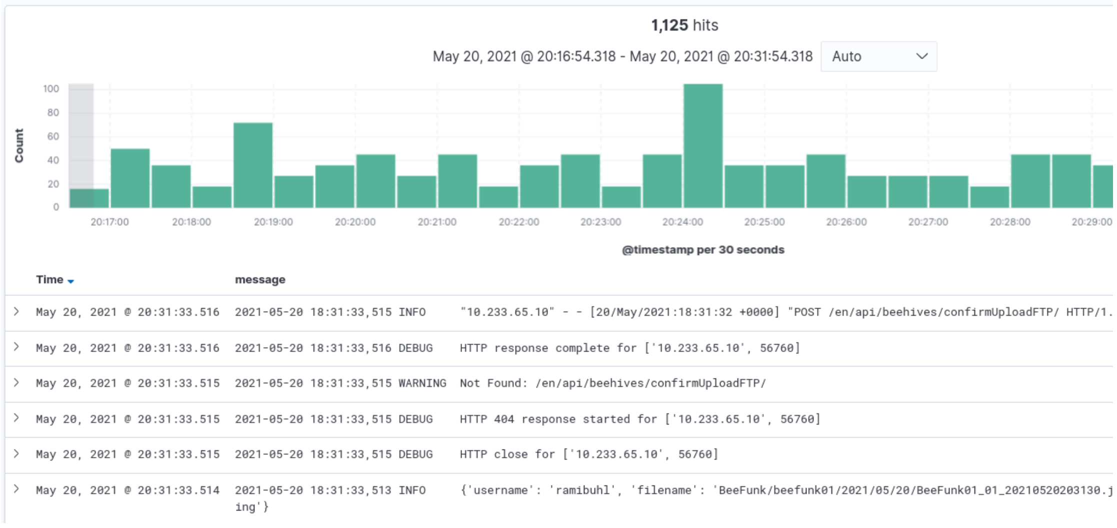
In summary, although the installation seemed simple at the end, we have spent a considerable amount of time trying to enable authentication for our logs. It started with unclear instructions on how to enable the security package for Elasticsearch, and why it was necessary. Then we had problems with certificate creation, and lastly we had to properly set the accounts for each component. The account configuration might sound simple, but there are many built-in accounts [9] for Elasticsearch. Moreover, as we were deploying all components with Helm, it was not clear which accounts we should use, or create ourselves, for log ingestion, communication, or user login in Kibana.

In Terms of actual usefulness of the log collection, it can be easily considered an indispensable tool. As discussed in the sections on common problems occurring in Kubernetes, logging is a very universal tool with a wide variety of benefits. Initially, we expected to use it in a few complex cases, but eventually, we never bothered to search logs using terminal tools.

One of the most important benefits is the full text search. Without a centralized logging solution like the EFK stack, the developers can at best use the *kubectl* to print logs. Printing the logs however, does not mean you can also browse them properly, and this requires the developers to somehow adapt. Thus, the developers are usually forced to use some workarounds to search through the logs, and they usually have to find a balance between functionality and accessibility.

For example, the developers are capable of moving the printed logs into a proper text search engine with various features. However, this is a bit time consuming, so they usually use a simple *grep* tool in the terminal. Using *grep* however, requires them to know its features, and regular expression patterns in order to fully leverage its functionalities. Which is not as easy as simply toggling a *Case insensitive* search option in some UI.

After the implementation, the developers are no longer forced into finding workarounds, as the most accessible solution is a full text search engine. In more concrete terms, as you can see in Figure 3.2, a single web application was able to produce a 1000 log entries in just 15 minutes. As you can imagine, searching through so many logs using tools with limited functionality, and basically no user-friendly UI, can take some time. However, after we have implemented the EFK stack, the developers do not have a lot of reasons



```
  Time        message
> May 20, 2021 @ 20:31:33.516  2021-05-20 18:31:33,515 INFO     "10.233.65.10" - - [20/May/2021:18:31:32 +0000] "POST /en/api/beehives/confirmUploadFTP/ HTTP/1.
> May 20, 2021 @ 20:31:33.516  2021-05-20 18:31:33,516 DEBUG    HTTP response complete for ['10.233.65.10', 56760]
> May 20, 2021 @ 20:31:33.515  2021-05-20 18:31:33,515 WARNING  Not Found: /en/api/beehives/confirmUploadFTP/
> May 20, 2021 @ 20:31:33.515  2021-05-20 18:31:33,515 DEBUG    HTTP 404 response started for ['10.233.65.10', 56760]
> May 20, 2021 @ 20:31:33.515  2021-05-20 18:31:33,515 DEBUG    HTTP close for ['10.233.65.10', 56760]
> May 20, 2021 @ 20:31:33.514  2021-05-20 18:31:33,513 INFO     {'username': 'ramibuhl', 'filename': 'BeeFunk/beefunk01/2021/05/20/BeeFunk01_01_20210520203130.j
                               ing'}
```

Figure 3.2: Chronological summarization of log amounts in Kibana.

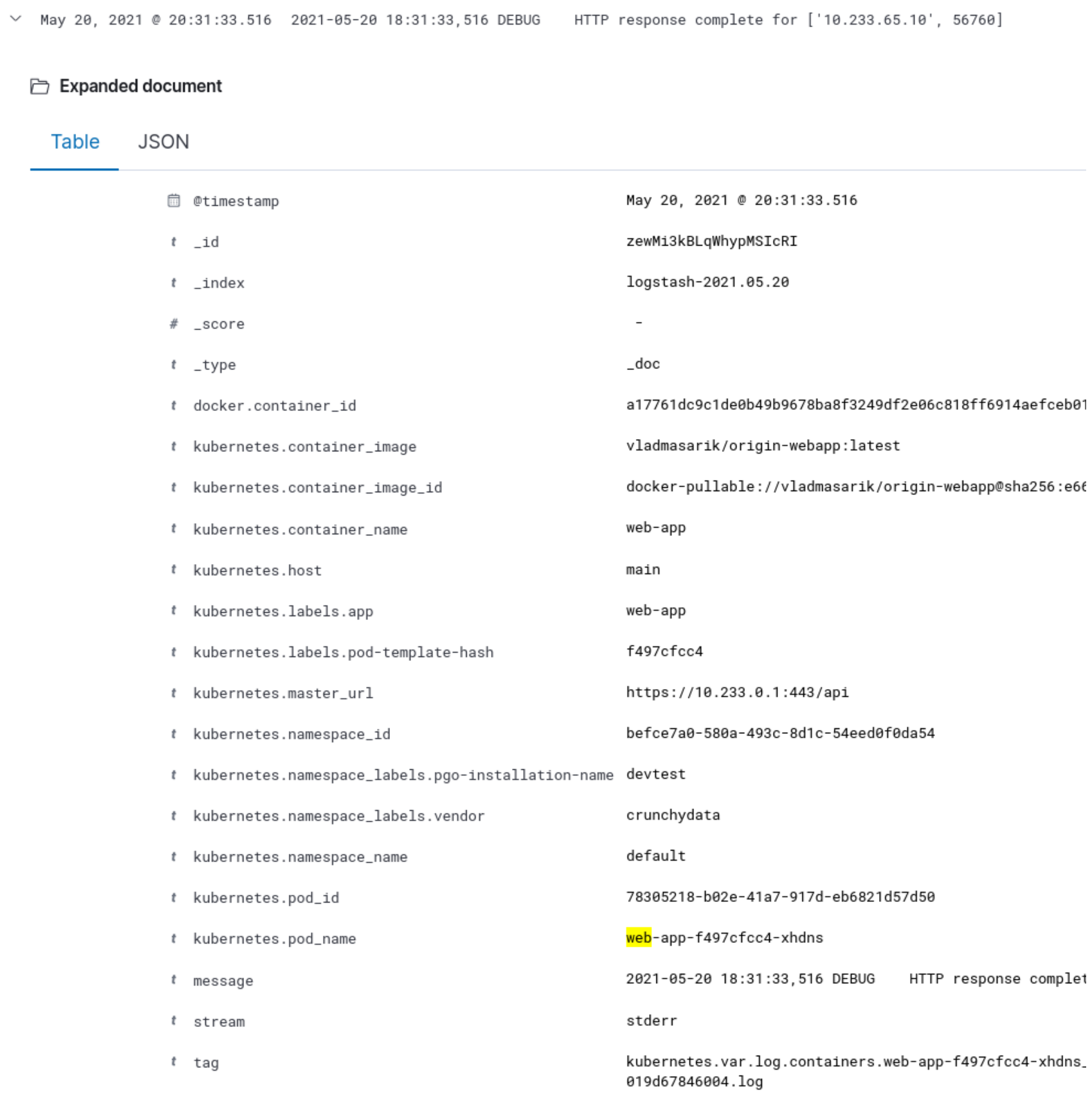
to keep printing logs and then searching through them in the terminal.

Yet another helpful feature that comes with EFK stack implementation are metadata. When Fluentd starts collecting the logs, it is also capable of adding various kinds of metadata in the Elasticsearch. You can see an example of the metadata that are added in Figure 3.3. These metadata are not always useful, however, many of them could be considered crucial, such as the Kubernetes pod names, or formatted timestamps. These two sources of information are especially helpful for developers, as they allow them to filter the logs based on information that is not available without the EFK stack.

Additionally, the timestamps are able to limit a time frame of the logs, and thus allow the developers to search for errors in multiple applications within a given time period. This allows you to decrease the amount of logs that have to be searched. Which might not seem to be a lot, but in cases where we have 10 application instances, each creating around 1000 logs every 15 minutes, and they run for 3 months, the logs grow to enormous sizes.

Moreover, if you combine the timestamp metadata with a search query for a word *error*, you are able to find all the applications that logged an error at that point in time. Therefore, the developers are able to limit the number of applications they have to debug using cross referencing in order to find the cause of those errors.

Furthermore, in case there are many clones of some application, such as our web application, the developers do not have to print logs for each application manually. Instead, all of the logs are already ingested by the Elasticsearch, and the developers have to only use wildcards to search through logs of all the application instances. Therefore, if the application is stateless, and user requests are hitting different application instances every time a new request is created, we can see the logs of each instance. Thus, no matter where the user's requests are being redirected to, we can always pinpoint the instance that contains the error logs, and use them for further debugging.



```
May 20, 2021 @ 20:31:33.516   2021-05-20 18:31:33,516 DEBUG    HTTP response complete for ['10.233.65.10', 56760]
```

Expanded document

Table JSON

📅	@timestamp	May 20, 2021 @ 20:31:33.516
t	_id	zewMi3kBLqWhypMSIcRI
t	_index	logstash-2021.05.20
#	_score	-
t	_type	_doc
t	docker.container_id	a17761dc9c1de0b49b9678ba8f3249df2e06c818ff6914aefceb01
t	kubernetes.container_image	vladmasarik/origin-webapp:latest
t	kubernetes.container_image_id	docker-pullable://vladmasarik/origin-webapp@sha256:e66
t	kubernetes.container_name	web-app
t	kubernetes.host	main
t	kubernetes.labels.app	web-app
t	kubernetes.labels.pod-template-hash	f497cfcc4
t	kubernetes.master_url	https://10.233.0.1:443/api
t	kubernetes.namespace_id	befce7a0-580a-493c-8d1c-54eed0f0da54
t	kubernetes.namespace_labels.pgo-installation-name	devtest
t	kubernetes.namespace_labels.vendor	crunchydata
t	kubernetes.namespace_name	default
t	kubernetes.pod_id	78305218-b02e-41a7-917d-eb6821d57d50
t	kubernetes.pod_name	web-app-f497cfcc4-xhdns
t	message	2021-05-20 18:31:33,516 DEBUG HTTP response complet
t	stream	stderr
t	tag	kubernetes.var.log.containers.web-app-f497cfcc4-xhdns_ 019d67846004.log

Figure 3.3: Detail of metadata for logs as displayed in Kibana.

Overall, there are many other helpful features that deserve a mention, but the full text search, and the additional metadata, are simply irreplaceable. They have saved us many hours of debugging, and ever since we have implemented the EFK stack.

As perfect as the logging solution might seem, there are a few limitations we would like to mention. The metadata that the Fluentd adds can be modified, and we are able to create our own labels, or parsing rules. The benefits are that we can extract data, such as error messages or error codes, from the logs. This allows us to create even more precise search queries, and summaries of the logs. The issue is that adjustment of the parsing rules is not simple, and requires quite a bit of effort to implement. Therefore, a lot of very useful functionality, but we are not able to leverage it without spending a lot of our time on it.

Moreover, the Kibana offers a lot of custom settings for searching as well. In our opinion the default values are user-unfriendly, and we always adjust them to our liking. This configuration is not saved throughout the deployments of the EFK stack. We were trying to find a way to automate the setup of these adjustments, but we were not able to find any easy solutions. Therefore, the maintainability of the EFK stack is not as low as we would like to, although this has to be done once every deployment, which is currently a rare occasion.

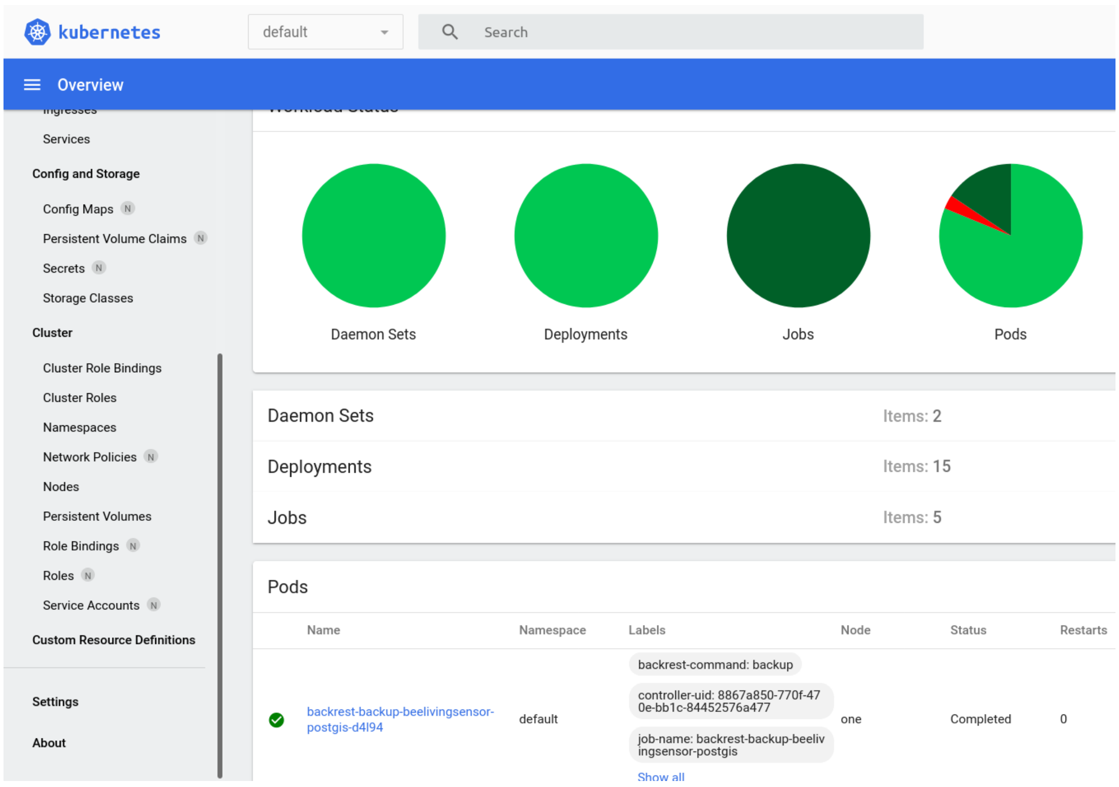
3.3 Kubernetes Dashboard

Kubernetes dashboard is a graphical web UI, which provides developers with a high level overview over the cluster, its events, state, and logs, as is shown in Figure 3.4. Moreover, it allows developers to not only view, but also create or modify the cluster resources such as services, deployments, or CRDs. It connects directly to the Kubernetes API, and fetches data about the different services. In general, it can be thought of as a *kubectl* replacement, although each of them have different advantages.

In terms of benefits that the dashboard has over the *kubectl*, one of its strongest sides is user friendliness. It is a Single Page Application, which improves its performance, and loading times rather drastically[42]. In the same way, it also truncates many of the Kubernetes API outputs. At a first glance it might not be considered useful, however, the amount of events, instances, and logs that are produced by the applications rises rather quickly. As such, there is a point in development, when trying to browse Kubernetes objects results in your terminal being filled with the command's output to the point where the first lines, that were printed by the same command, are lost.

With *kubectl* you would have to also *build* more complex commands. As an example, if you wanted to execute a command inside a pod, you would have to find out name of the pod with one command, find name of the container in that pod with another, and then write the command for execution:

```
$ kubectl exec -it prom-production-axymss2g --container envoy-
   proxxy -- ls -l /
```



Daemon Sets	Items: 2
Deployments	Items: 15
Jobs	Items: 5

Pods

	Name	Namespace	Labels	Node	Status	Restarts
✓	backrest-backup-beelivingsensor-postgis-d4l94	default	backrest-command: backup controller-uid: 8867a850-770f-470e-bb1c-84452576a477 job-name: backrest-backup-beelivingsensor-postgis Show all	one	Completed	0

Figure 3.4: Home view of the dashboard showing status of different services, and other information.

This is not difficult, however, in the web UI all of the required information is already displayed. What is more, the web UI executes these commands when a developer clicks on a link. Thus, the workflow would consist of clicking on a pod name, then container name, and lastly on a "*Execute in a pod*" icon, which is much faster than writing all the commands manually.

Additionally, some tools such as *cert-manager*[15], add Custom Resource Definitions (CRDs), which are developer-created Kubernetes representations of applications. They allow the developers to leverage Kubernetes management functionality, to manage their own applications. Unfortunately, during tool installations it is not clear which custom objects are created. Therefore debugging applications that use them is rather difficult, as we do not know where the logic of the application is stored.

Lastly, outside of user friendliness, the dashboard UI is especially useful when the developers do not have an external access to your platform for security reasons. To execute *kubectl* you would have to be either inside the cluster with *ssh* or an external Kubernetes API would have to be exposed. However, the web UI allows developers to specify access rights in a more detailed fashion, and since the dashboard runs in the cluster, Kubernetes API does not have to be exposed, thus lowering the possible attack surface. There are many more smaller *quality of life* improvements that the Kubernetes dashboard introduces, however listing all of them would be too long.

In terms of effort that we had to spend on running the dashboard, the set up was not straightforward, as there was a slight lack of documentation. We used Helm to deploy the dashboard, however by default the dashboard accepts only HTTPS traffic, and creates its own certificates. In our case though, all of the traffic within the cluster is HTTP only. Thus, when our proxy redirects external traffic into the dashboard, the user is denied access with a 400 error, and no further explanation. It took us quite some time to find the reason for why our proxy server was not using the HTTPS, and used only HTTP, which caused the dashboard to return an error. Unfortunately, the only hint we found was through *curl* logs that you can see below:

```
* Issue another request to this URL: 'https://dashboard.
   beelivingsensor.eu/'
*   Trying 52.188.109.11:443...
* TCP_NODELAY set
* Connected to dashboard.beelivingsensor.eu (52.188.109.11) port
   443 (#1)
* ALPN, offering h2
* ALPN, offering http/1.1
* successfully set certificate verify locations:
*   CAfile: /etc/ssl/certs/ca-certificates.crt
  CApath: /etc/ssl/certs
...
*  SSL certificate verify ok.
> GET / HTTP/2
> Host: dashboard.beelivingsensor.eu
```

[15]https://cert-manager.io/

```
> user-agent: curl/7.68.0
> accept: */*
>
< HTTP/2 400
< date: Mon, 11 Jan 2021 13:32:12 GMT
< strict-transport-security: max-age=15724800; includeSubDomains
<
Client sent an HTTP request to an HTTPS server.
* Connection #1 to host dashboard.beelivingsensor.eu left intact
```

As you can see, the error "*Client sent an HTTP request to an HTTPS server.*" states that the dashboard received HTTP request, although at the beginning we are using HTTPS "*Issue another request to this URL: https://dashboard.beelivingsensor.eu/*". Something changed the request, but we did not know what. After some time, the proxy server logs provided the answer:

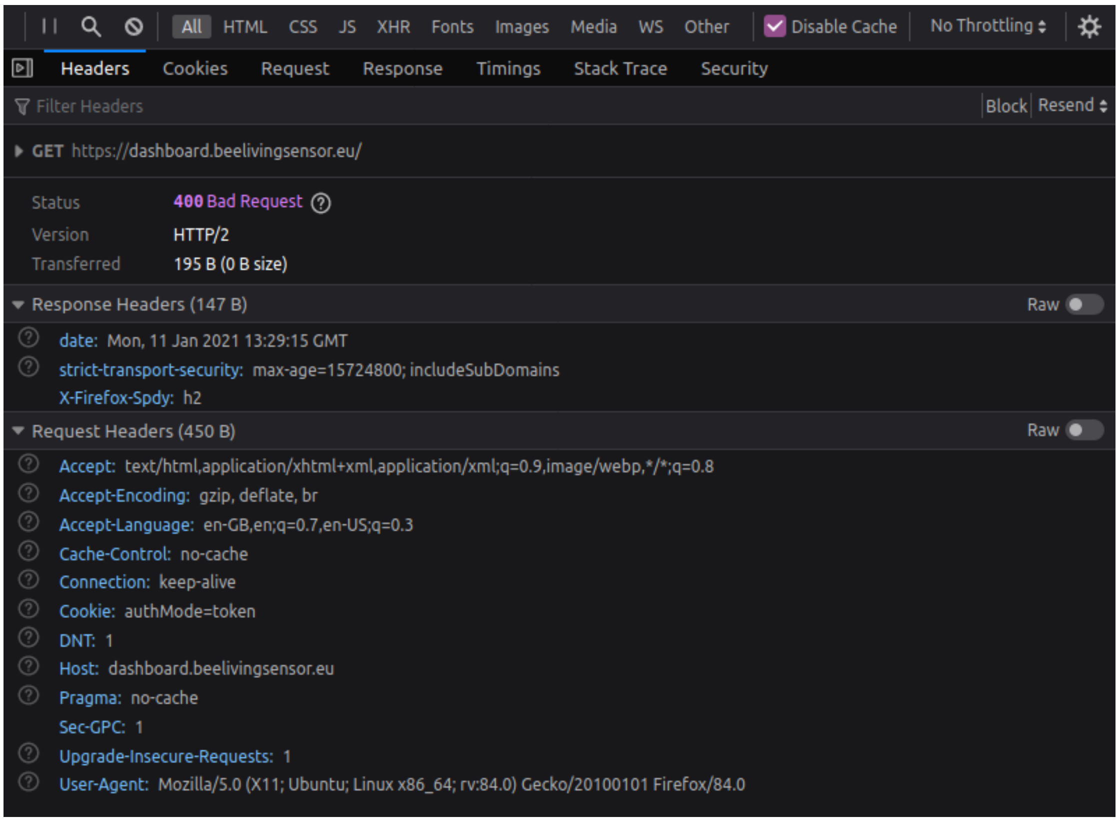
```
2021/01/11 13:36:34 [error] 1178#1178: *256996 recv() failed (104:
     Connection reset by peer) while sending to client, client:
    194.230.155.207, server: dashboard.beelivingsensor.eu, request:
     "GET / HTTP/2.0", upstream: "http://10.233.65.13:8443/", host:
     "dashboard.beelivingsensor.eu"
194.230.155.207 - - [11/Jan/2021:13:36:34 +0000] "GET / HTTP/2.0"
    400 48 "-" "Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:84.0)
    Gecko/20100101 Firefox/84.0" 245 0.002 [default-k8s-dashboard-
    kubernetes-dashboard-443] [] 10.233.65.13:8443 48 0.004 400
    c0c929b37be6e8cea4353b69147db57f
```

Arguably, it is hard to spot among the other logs, and even within the log message itself. However, you can see the request is sent to "*upstream: http://10.233.65.13:8443/*", which is our internal IP address of the dashboard service. Thus, at some point the proxy changes the HTTPS requests into HTTP.

In the end we were not able to find the cause. We had concluded that the effort required to set up the dashboard, and the proxy to accept HTTPS requests would simply outweigh the benefits of using HTTPS for traffic between the proxy and the dashboard.

As such, we have aimed to configure the dashboard to accept HTTP traffic. The documentation specified how to enable it, but there was another problem. Allowing HTTP traffic disabled dashboard authentication, and anyone could access the dashboard externally. This was of course a security risk. After some research however, we were able to solve this issue as well, and the dashboard authentication for HTTP requests was enabled.

Yet another problem with the dashboard deployment configuration was that it allowed for its own creation of ingress rules automatically. This could be useful in some cases because our ingress rules list would get smaller, and we would probably never have to worry about managing it in the future. It would be managed by the community, and arguably most of the time functional. The problem was that if we were to change the ingress rules, for example changing the domain name, then we would also have to keep in mind the extra dashboard configuration hidden in Helm charts. Although changing



```
GET https://dashboard.beelivingsensor.eu/

Status        400 Bad Request
Version       HTTP/2
Transferred   195 B (0 B size)

Response Headers (147 B)
date: Mon, 11 Jan 2021 13:29:15 GMT
strict-transport-security: max-age=15724800; includeSubDomains
X-Firefox-Spdy: h2

Request Headers (450 B)
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Encoding: gzip, deflate, br
Accept-Language: en-GB,en;q=0.7,en-US;q=0.3
Cache-Control: no-cache
Connection: keep-alive
Cookie: authMode=token
DNT: 1
Host: dashboard.beelivingsensor.eu
Pragma: no-cache
Sec-GPC: 1
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:84.0) Gecko/20100101 Firefox/84.0
```

Figure 3.5: Kubernetes dashboard request returned 400 error, but without any reasoning, or helpful information shown in browser.

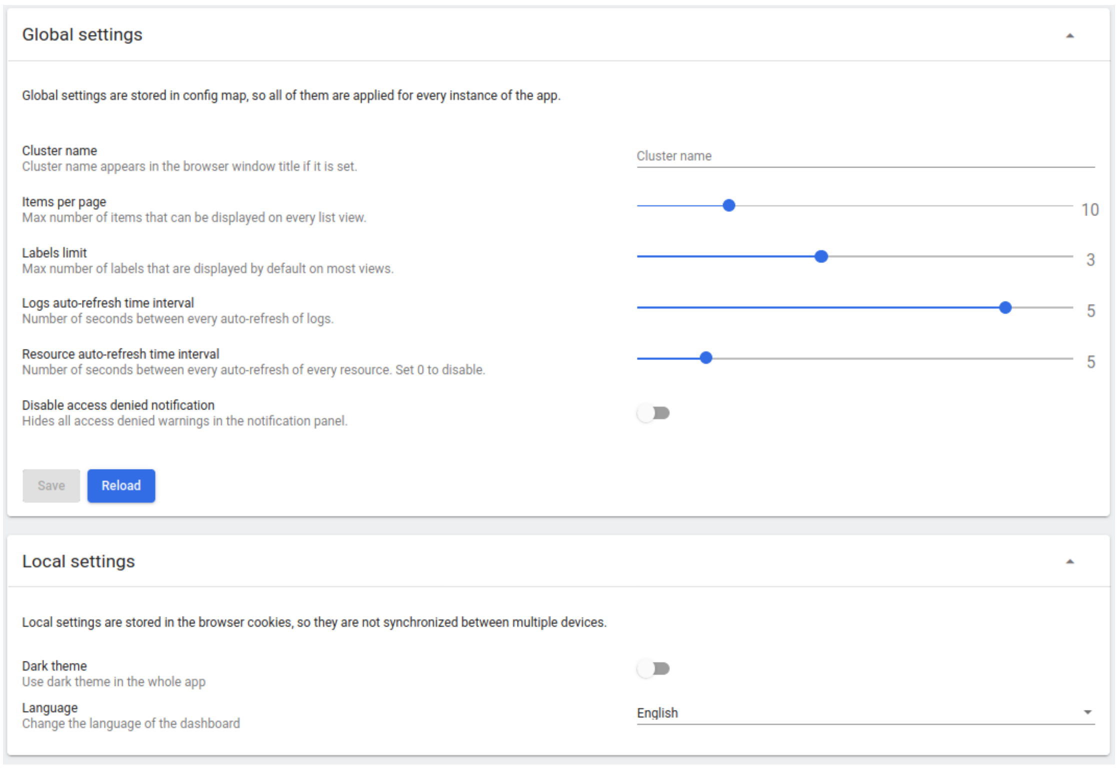
ingress rules is rather rare, it was quite probable in our case, as at time of writing, we have encountered problems with changing domain names. In the end, we have decided to keep all of the rules in a single configuration file, and we used our own ingress rules.

Overall, we have spent most of our effort on debugging the 400 request error, and finding the correct log error within the proxy server logs. We had hoped that web browsers would provide us with helpful information, however as you can see in the Figure 3.5 that was not the case, and we were only able to find the error through usage of *curl*.

As for the limitations, the dashboard provides an informative overview, but it will not replace the *kubectl*. We believe they are complementary to each other. The dashboard streamlines many very simple and common tasks, but it lacks flexibility in the area of customizing the output and modifying Kubernetes resources.

Moreover, the amount of dashboard settings is limited as can be seen in Figure 3.6, which prohibits us from customizing it to our needs. The configuration is also not saved throughout deployments, and it would require us to manually readjust them every time, thus it possibly even increasing maintainability.

Additionally, when a developer wants to log in, they have to generate a *bearer token*,



Global settings

Global settings are stored in config map, so all of them are applied for every instance of the app.

Cluster name Cluster name appears in the browser window title if it is set.	Cluster name
Items per page Max number of items that can be displayed on every list view.	10
Labels limit Max number of labels that are displayed by default on most views.	3
Logs auto-refresh time interval Number of seconds between every auto-refresh of logs.	5
Resource auto-refresh time interval Number of seconds between every auto-refresh of every resource. Set 0 to disable.	5
Disable access denied notification Hides all access denied warnings in the notification panel.	

Save Reload

Local settings

Local settings are stored in the browser cookies, so they are not synchronized between multiple devices.

Dark theme Use dark theme in the whole app	
Language Change the language of the dashboard	English

Figure 3.6: Kubernetes dashboard containing a very limited number of adjustable settings.

which is essentially a key proving their identity. This requires the developer to connect to the cluster, and print out an admin secret. It is not a complex operation per se, but it is time consuming. The dashboard session also expires after an hour of inactivity, which requires authentication using the *bearer token* again. Thus, might lead to a rather cumbersome experience for developers.

3.4 Health Service Monitors

As part of our Failure of External Services section 2.5, we have identified that failure of external services is a common problem. We have also described how we are able to mitigate these issues for each of the services that can be affected. The analysis showed that although we are using multiple external services, some of them are very crucial, and this section aims to help with their debugging.

Specifically there were two services, the DNS provider Hostek and the risk that the DNS resolution could fail, and the Azure platform on which we host all of our infrastructure. This includes storage, network, load balancers and other computing resources. These resources are crucial, and apart from logs in our applications showing that some of the external APIs are suddenly unreachable, we are not able to properly debug situations where our providers would experience outages. The developers would be suddenly met with unresponsive or missing resources, and without any explanation for what might be causing the issue.

We have analyzed the providers, and found out that they allow their users to subscribe to notification about potential outages. Thus, we believe that if developers would be met with a sudden rise in failures on our platform, or perhaps if our services were unreachable, the developers would be notified of possible issues with the external services we use. We believe that this will help developers to faster assess the situation without the need to search the logs of applications, and possibly having to deduce the underlying problem.

Afterall, if multiple services stop responding, and there were no recent changes to the platform, the developers can easily assume that they have simply missed the change done to the platform, and continue searching for this non-existent platform change. However, with a notification in the form of an email, SMS, or desktop push notification it should be hard to miss that the root cause is not the platform, but an outage in the underlying infrastructure.

Overall, there was no need to search for alternatives to notifications, as in both cases of Azure infrastructure, and the DNS provider, they provided us with the exact features we needed in a very simple manner. Set up of the notifications took barely 20 minutes.

For Azure, we have created notification alerts for any network, storage, load balancer, or compute resources issues using the Azure Health service[16]. We will receive notifications for issues in regions of East US, and South-East US, which is where our Kubernetes cluster and object storage are located respectively. Any developer can later provide their preferred type of notifications, and subscribe to this service. Currently we have set up

[16]https://azure.microsoft.com/en-us/features/service-health/

email notifications.

For our DNS provider Hostek, anyone interested in their status updates can subscribe to their RSS feed[17]. The developers can use any RSS feed reader according to their preference. There are unfortunately no customization options for which region to receive notifications, but we believe this is a very minor concern.

Since the implementation of the notifications, we were fortunately not having any problems with the underlying infrastructure. As such we are not able to fully assess how useful they are, however again, we believe that this decreases the time to check whether the Azure platform or the DNS resolution outage might be the cause of our problems.

Overall, there are only two limitations to this solution which we were able to find so far. First, there might be multiple ongoing outages happening in Azure and Hostek, however that does not mean we are automatically influenced by them. The outages might be simply happening in sections which we are not using. Therefore, there are sometimes false positive notifications, but this is not a problem if we are able to verify that our platform works as intended, and we can simply ignore the notifications.

Second, although there might be problems with accessing our platform, and there are clear indications that there is currently an ongoing outage with DNS or Azure, it does not necessarily mean that the outage is the root cause of our failure. This should be a rather rare occurrence where both our external services and our platform experience problems simultaneously, but it can happen, and it might lead to our developers thinking the issue will resolve itself. However, that will not be true, and we will be simply waiting for a fix that will never come. This can be very damaging to the comfort of our users, and we are not sure there are any solutions for such situations.

3.5 Alerting

In order to answer our RQ 1.2. and RQ 3, we have looked at the ways to detect anomalies on our platform, and the possible ways of alerting the developers in case of said anomalies. As discussed in the previous section 3.1, while trying to help developers debug problems faster, we have also stumbled upon a tool capable of notifying the developers in case of problems.

This tool is part of the Prometheus, and it is referred to as Alertmanager. Although somewhat counter intuitively, the Alermanager itself does not take care of triggering notifications for developers. It is responsible only for the management of the already incoming alerts in the form of API requests, and for sending out notifications in the form of SMS messages or emails. The reasoning behind this architectural decision is that the developers might want to define many unexpected situations that they would like to be notified about. These definitions are not handled by the alertmanager, but by the Prometheus alerting rules, and its benefits are described below. Thus, the whole solution that allows us to alert the developer in case of platform issues is consisting of three components within the Prometheus.

[17]https://hostek.com/serverstatus/rss.php

The first component, which was already described in section 3.1, is simply a collector of the metrics. For example, if we want to observe resources such as disk utilization, or free memory, this collector takes care of it. The collected data is then saved into a Prometheus database for later usage.

Second component are the Prometheus alerting rules. These are simply represented as configuration files, and should describe situations when an alert should be triggered. They are defined by developers, and contain information, such as at what CPU usage should an alert be triggered, what is the severity of the situation, and description of the problem. These alerts are then sent to the Alertmanager for further processing.

The last component is the Alertmanager. As we have already mentioned a bit, Alertmanager is responsible for *managing* these alerts. By managing we mean actions such as grouping the alerts, silencing them, repeating, or postponing. The reasons are that in case the developers create many definitions of problematic situations, for example when there are only few MBs of memory left, or when pods are reporting crashing, all of these alerts would be triggered simultaneously, and they would easily overwhelm the developers with notifications.

Thus, the Alertmanager allows defining rules, not to send 15 emails because 15 services are failing, but only a single email describing that those 15 services are failing. In a similar fashion, there are cases when we are not able to handle errors, and we have to wait for them to be automatically resolved. The time until everything is fixed might be unknown to us, and it would be unnecessary to receive an email every minute, while we wait for the issue to be resolved. Therefore, the Alertmanager offers us the possibility of temporarily disabling or silencing the notifications, and modifying the repeating intervals of the notifications in case the issue is still present.

We are also able to set a waiting period for the notifications. This would be helpful in cases where we monitor the CPU usage, and it might have a temporary spike. Short bursts of CPU utilization are not necessarily causes for concern, but by default we would receive a notification each time there would be a small spike. Fortunately, we are able to tell the Alertmanager to notify us only if the CPU is at more than 90% for at least 10 minutes, which might be a good reason to preventively investigate the situation.

We have already outlined a few benefits of alerting, but there are many more. Although the Prometheus is able to collect only numeric data, it is still possible to encode the status of the individual pods as numbers. This includes status types such as *crashing* in case the main container process finishes with non-zero status, *pending* for when we are trying to deploy new versions of our applications, but they cannot start, or *not ready* in cases where the pod started successfully, but it is not ready to receive any requests. These are probably the most frequent states, but Prometheus can have alerting rules for any other states available.

Moreover, as we have mentioned in section 3.1, there is a possibility to create our own metrics. These can be then collected by Prometheus, and we can easily write our own alerting rules for them. One example which we would like to implement in the future is the number of connected cameras to our FTP server. There are cases when the cameras might stop uploading new videos. This is hard to spot as currently we are uploading more than 150GB of videos per day, and ensuring that we have received videos for each day, and for each camera manually is rather impractical. Therefore, we would like to create a metric which could notify us about cameras that might be experiencing problems. At the moment though, we do not have this metric, and so detecting issues with the cameras is rather difficult.

With the implementation of alerting, we are also able to monitor the HTTP traffic coming to our website. Currently, we have not defined these rules, but finally we have the possibility to be notified about responses that indicate errors. Throughout the development, many parts of the software are changed. Some of the changes might include bugs though, and we are not always able to catch them during testing. Now however, we are able to notify the developers about the increased number of responses with HTTP error code 500, which indicates error in our servers. This helps us to find mistakes, which we were not able to find during testing, and possibly fix them before other users encounter them.

We have tried to look for a few alternatives for alerting. In a similar fashion to the alternatives in the PG stack, we have managed to find quite a few of them, although many of them were purely commercial. Therefore, we have not considered them, as we would have to first buy them in order to assess their features and the effort required to integrate them with our platform. To our surprise the number of free and open source alternatives which matched our main candidate Prometheus was even lower than with monitoring.

We have come across the Wavefront[18]. In their documentation they showed that it was possible to integrate it with Kubernetes through Helm, however to the best of our knowledge we were not able to say whether this project was open source. There are a few projects they have published, but their documentation suggests that we would need accounts for their platform in order to implement notifications for our developers.

We have also discovered the Cabot[19]. Their documentation mentions that although the tool is in wide usage, they have stopped maintaining the project. This disclaimer was now present for over a year, and as such we have decided not to consider it. Similarly, we have discovered the Nightingale[20]. This project was unfortunately unusable as well, as their documentation is offered only in Chinese language, which could pose problems with maintenance.

After implementing Prometheus with ease, we were also expecting the same for the alerting, but this was not the case. The first difficulty we encountered was an unclear distinction between the alerting system provided by Prometheus, and the Alertmanager. For a while we have considered both of these components to be a single entity, and we thought that the documentation considered them to be interchangeable concepts, which unfortunately, cost us quite a bit of effort.

Second, we have had quite a few difficulties with the configuration of the alerting rules, but also with the Alertmanager's notification system. Since we have deployed the Prometheus using Helm, the Alertmanager had to be deployed in the same fashion. We

[18]https://docs.wavefront.com/index.html
[19]https://github.com/arachnys/cabot
[20]https://github.com/didi/nightingale

had assumed that even though the deployment process was different, the configuration would not change much from the structure they have described in their documentation.

This assumption turned out to be incorrect after we learned more about the Alertmanager. We have discovered that the Prometheus version that was deployed using Helm used a different kind of configuration architecture. It was still possible to pass the configuration to Prometheus and Alertmanager via configuration files and even through command line options, but this was very hard to maintain. Instead, both applications were configured through CRDs that represented their configuration.

Furthermore, we have encountered other problems with the configuration because the documentation for these CRDs was inconsistent. For example, there were multiple ways of configuring notifications through emails. Some parts of the documentation mentioned that a few of those configuration options were deprecated, and were kept functional only for backwards compatibility reasons, but it was not clear which parts were meant to be used.

Lastly, both Prometheus and Alertmanager included a few default configurations for alerts, and notifications. We are assuming that under normal circumstances, these alerts would work as intended. That is notifying the developers about some Kubernetes related pods not existing. However, in our case the alerts were expecting these pods to be running in our cluster, but they were not. We have not managed to understand why Prometheus was expecting these pods to be present, or why they were missing from our platform. In the end, we had to manually disable these alerts within our notifications, as otherwise we would be constantly receiving emails reporting those pods to be missing.

After we understood the documentation better, and explicitly disabled the alerts, the setup was straightforward. The Prometheus required CRDs of type *PrometheusRule*, which specified situations when it should create alerts. While the Alertmanager required CRDs of type *AlertmanagerConfig*. There, we have specified the notification type, and any extra details it required. At this point it is up to the developers to decide which notifications they prefer, but so far, we have created email notifications for any alert that Prometheus might trigger.

The email notifications however required further work from our side. In order to send emails from Alertmanager, we have to specify an SMTP server. This server acts as an entry point for our email, and it takes care of properly routing it to our desired target. Unfortunately, since it is easy to create emails, and spam people with them, many SMTP servers block any email traffic that is not somehow authenticated[14]. Google provides this SMTP service to anyone with a Google account so we have created one, and after solving a few issues with authentication, we were able to successfully send emails triggered by Prometheus.

After the successful integration of the Prometheus alerting rules, and the Alertmanager configuration we have started noticing the benefits fairly quickly. As the development of our platform, and the applications it hosts, continued, a new code was uploaded into our code repository on a regular basis. This code was manually tested, and we made sure to only commit working code. Therefore, we were expecting the automatically deployed applications to work as well. However, there were times when we have uploaded code that failed to run due to various reasons. As this new functionality was built into our application, and deployed onto our platform, we were however quickly informed that our applications were unable to start because of the undetected bugs. Thanks to the alerting system though, we were quickly informed about our mistakes, and started fixing them.

Another situation where the alerting helped us avoid problems was while improving the Kubrnetes cluster. We were trying to debug a few issues with the network. In order to debug it, we had to manually download and execute a few Docker images. These images were not necessarily large in size, but combined with the images already stored on the VM, the metrics and logs collected over the previous months, and other application data, they filled up the disk. This quickly triggered Prometheus alerts warning us that our disks were nearing their capacity. We have immediately noticed this, and mitigated the issue, but we believe that without those warnings, our applications would stop functioning properly at some point. Thus, we were able to prevent possible future problems. This could have resulted in our databases having no space to save their data, and our *model* application also needs space for downloading video for further analysis. Ignoring such an issue would easily lead to our platform not operating properly.

We have also successfully received notifications about crashing applications. However, we were also directly working with the platform in those cases. In other words, the developers were causing the crashes unintentionally, but we were already aware of these problems even without the emails. At least it still shows that in case the applications unexpectedly stop working, we would be promptly notified.

The Alertamanager helps the developers in many aspects, but there is still one limitation we would like to mention. There are some events which cannot be monitored with Prometheus. The most important of them are the application exceptions. In cases where our applications throw exceptions, the server does not automatically stop, but rather it continues accepting the HTTP requests. Therefore, if the users trigger errors in our applications, we are not always able to notify the developers of these errors. Fortunately, during our research we have noticed that our log collector EFK stack should be able to provide this functionality, although so far we have not explored this solution further.

3.6 Kubernetes Self-healing

Kubernetes is meant to act as an automatic application management platform. Part of those management responsibilities are its *self-healing* capabilities, and thus, it was designed with them in mind. *Self-healing* is a rather sizable topic in Kubernetes, but it means that although the applications that are running on the platform are unexpectedly terminated, Kubernetes will restart them if possible[15]. Therefore, any issues that might cause applications to stop, for example hardware failure, memory leaks, application errors, networking issues, or the need to adjust pods, will be automatically mitigated, and this mitigation will allow the applications as a whole to continue working.

The obvious benefit of this is the additional robustness for our applications. The applications are updated on an almost constant basis, and even though we might be using best practices for testing, there might be some bugs that we have simply missed. In practice this means that the application could be working for a while, but at some point it will encounter an error. More often than not, this error is unexpected, and it causes the application to exit unexpectedly.

There are a few workarounds even for applications that are not containerized, for example you can create a service definition in your OS, and in case of a crash, the process will be restarted. However, this workaround is not always easy to implement. In the case of Kuberenetes though, this functionality is a default behavior for most of the containerized applications, although not all of them.

Moreover, if you were to define a service in your OS, it would not always meet the non-functional requirement of availability. Which, compared to service availability that Kubernetes provides, is achievable in a much easier way, although not by default. In any case, if configured properly, Kubernetes enables the availability by running multiple copies of your applications. Therefore, even if one of the copies stops working because of an error, there are still other applications running, which can replace the terminated application. This, combined with the robustness that can be provided by Kubernetes, allows developers to write stateless applications that can achieve very high levels of availability. Therefore, lowering the requirements for maintenance even in case our applications contain errors.

Furthermore, hardware is bound to fail at some point, and the cloud providers operate on such a large scale that you never know when your VM might simply cease to exist. Again though, self-healing properties of Kubernetes can provide some help. In situations where you VMs, and pods running on those VMs stop responding, Kubernetes will mark them as unavailable. Then, in a similar way with crashed applications, Kuberentes will start new clones of the application as necessary. Compared with the traditional services running directly on top of an OS, if the OS would fail there would be no attempt at restoring the applications, and they would stay unavailable.

As we have already outlined a bit, Kuberentes comes with the self-healing functionality already implemented, but for it to function, it requires developers to properly configure it. Although we were expecting the set up to be difficult, we were surprised at how relatively simple it was. As explained further below, we only had to create deployment configurations for each applications the we were using.

The lowest representation of an application in Kubernetes is a pod, which is a collection of containers that live and die together. Pods themselves however, do not have this self-healing capability. If a single pod dies, Kubernetes will delete it, and never restart it. The developers can however create deployments, which can be viewed as an abstraction of an application. These deployments can then specify how many clones of the applications should be running at any point in time, and it is these deployments that actually provide the added benefit of robustness and high availability.

It should be noted however that creation of the deployments requires some level of contextual knowledge. The developers are able to specify the number of applications running in parallel. Of course configuring deployments to run only one application in parallel nullifies the benefits of high availability, but running 10 or 50 of them is not advised either. In the end, it all depends on the hardware resource requirements of your application, and on the capabilities of VMs that you have access to. To a certain degree this can be solved automatically, and it is actually described in further detail in section

3.7.2. However, in the end, the developers are needed in order to encode their contextual knowledge of their present situation into the deployment's configuration.

After implementation the benefits materialized very quickly. The self-healing capability was especially useful for cases when we were developing new functionality for our website. Many times we have managed to write code that would work locally, but then we would forget to adjust configuration for our production environment. Thus, whenever we deployed a new version, it usually failed to start, but after adjusting the environment we just had to wait for Kubernetes to restart the pods. This helped the developers avoid having to manually restart the failing applications, and thus saved their time.

Unfortunately, at some point in time, one of our distributed database instances failed to communicate with the rest of the database cluster, and so other databases marked it as a backup instance, which should be used for read only operations. However, this database was still simultaneously somehow labeled as a primary source of information as well. The results were that when our applications tried to save data into it, their requests were denied because only read-only operations were allowed. It took us some time to understand why this was happening, but after we have identified the issue, the solution was simply to delete the database pod. Because of the deployment's self-healing properties, the database instance was simply restarted after the deletion, and this time worked without a problem. If we did not have the database deployment configured, we would have to manually recreate the whole database, which would be a rather time consuming task.

The only limitation which we were able to notice was that we had to create configurations for each application individually. Currently, we have set up deployment configuration for every application we have developed so far, but any newly introduced applications will require developers to create their corresponding deployment configurations.

3.7 Automated Scaling

In order to support developers in the future, and lower the requirements for maintainability, we have looked at the autoscaling features of Kubernetes. We hope that these features will free the developers from the need to manually scale, and configure newly created VMs and pods.

There are two main parts to the whole autoscaling concept. These parts can easily exist on their own, however, together they bring much more value in terms of improving the maintainability of our platform. These parts are the Horizontal Pod Autoscaler[21] (HPA) and Cluster Autoscaler[22] (CA). The two components are rather complex and need a more detailed description, as each has its own benefits and limitations. Therefore, they are further described in their respective subsections.

[21]https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/
[22]https://github.com/kubernetes/autoscaler

3.7.1 Horizontal Pod Autoscaler

As the name might suggest the Horizontal Pod Autoscaler (HPA) is responsible for increasing or decreasing the number of running pods. In the previous section 3.6, we have described what a deployment is, and how it is capable of maintaining a certain amount of parallel applications running through its self-healing mechanisms. In a similar fashion to the deployments, HPA comes as a part of Kubernetes, but it requires further setup in order to function. After properly executing the HPA, it is able to automatically control the deployments, and dynamically adjust the number of parallel applications that should be running at any point.

The HPA adjusts the number of copies based on the hardware resources the application uses. Although we have already integrated Prometheus into our platform, and we are able to monitor such metrics, the HPA requires a different source of information. This source is basically just another service running on Kubernetes, and it is called *metrics-server*[23]. Overall the metrics collected by the *metrics-server* are similar to the ones collected by Prometheus, however they come from a slightly different source, and are automatically visible to the HPA.

The HPA can then use, for example the memory usage metric, to decide that some of the parallel instances of some application, which are referred to as pods, are using their maximum assigned memory capacity. Then based on this information the HPA decides that there should be more pods running, as it assumes that adding more application instances will help even out the load on the pods, and average memory usage per pod will then decrease. In a similar way to increasing the number of pods, if the HPA sees that the memory usage is on average low for longer periods of time, it will decide to decrease the number of pods, as some of them are not being used fully, and the present load could be handled by the rest of the pods.

The benefits of HPA come from the assumption that our applications are properly written microservices. Therefore, contrary to monolithic applications, we are able to scale only the parts of our web platform which are being used. In other words, if we had a monolithic application, and we would want to scale it in order to meet increased user requests, we would have to create a copy of the whole application. This would also duplicate parts of the application that are not being used frequently, thus consuming more hardware resources than necessary. While the idea of correctly designed microservices is that we would scale up or down specific functionality of the whole application, based on the needs of users. Therefore, we would not be consuming resources that are not needed, and it would allow us to be much more flexible in terms of handling specific user requests.

The other benefit is of course that the application will be scaled according to the current needs. Therefore, developers will not be required to periodically monitor application needs, and then manually adjust the number of running pods in order to meet the demand, which is fairly impractical. Especially if you consider that some applications might be more used during the day, for example the website, while others can be given more resources during the night, for example the video analysis.

[23]https://github.com/kubernetes-sigs/metrics-server

Lastly, the HPA works very well with the Cluster Autoscaler (CA) component, which is described in the subsection 3.7.2. The HPA itself can create new pods, but it cannot create more of them than what the current hardware allows. Therefore, we are limited by the number of VMs that are running, and we would be forced to manually create new VMs in order to handle more user requests. Similarly, if we had many VMs running, but the incoming traffic would be low, the HPA would scale down the number of pods, but we would be still paying for the infrastructure, which is basically not used, as we no longer need to process large amounts of user requests.

The integration of HPA seemed simple, but it turned out to be a rather complex endeavor. The first step we had to follow was to implement the already mentioned *metrics-server* application into our cluster. Because it is used to collect data about the resource usage of our cluster, it needs to communicate with the main Kubernetes process called *kubelet*. The communication is executed through an HTTP API that *kubelet* exposes, and of course in order to have this communication secure, we need to use HTTPS, which requires certificates.

The first problem we have met with was the insufficient documentation for *metrics-server*. It was not clearly explained how to properly use the certificates, but after further research, we have managed to find a way to specify which certificates should be used. Unfortunately, the certificates we used were somehow not functional. We have spent a considerable amount of time trying to find the errors in our certificates, but in the end, the problems were in the incoming certificates from the *kubelet*. Below you can see output from a curl command:

```
$ curl -vvv -kI https://10.0.1.5:10250/stats/summary?
   only_cpu_and_memory=true
...
* SSL connection using TLSv1.3 / TLS_AES_128_GCM_SHA256
* ALPN, server accepted to use h2
* Server certificate:
*  subject: CN=main@1621337481
*  start date: May 18 10:31:21 2021 GMT
*  expire date: May 18 10:31:21 2022 GMT
*  issuer: CN=main-ca@1621337481
...
```

The URL that we have called was copied from the logs of the *metrics-server* application, which claimed that it was not able to match the certificate subject name with the expected IP address. As you can see from the *curl* output, this was the case because the certificates used the subject name *main@1621337481*, while the *metrics-server* looked for the *10.0.1.5* subject. Of course this would not work. Except that the documentation specified that if the certificate subject does not specify an IP address, it should look for a hostname. The hostname is clearly there, but the application somehow did not check for it. We have eventually managed to work around this issue by specifying that the *metrics-server* should not match the IP address, but hostname only.

However, we have hit yet another problem where the *metrics-server* refused to accept the certificates, even though it did match the subject. The reason was that they were self signed. This was understandable because our cluster does not use any external certificate authority to sign the certificates, and they are automatically created by the Kubernetes when the cluster is created. What perplexed us was why the *metrics-server* did not trust the certificates, which were signed by Kubernetes. We have provided it with the issuer's certificates, so it should not have problems.

The thing to notice in the logs is that the certificates specified *main-ca@1621337481* as an issuer of the certificate, and we have never experienced such a certificate issuer name before. To the best of our knowledge, all certificates related to *kubelet* always use the issuer name *kubernetes*. The solution therefore is to find the certificates for the issuer *main-ca@1621337481* so that the *metrics-server* would trust them. We have searched as many places as we could for these certificates, but we were unable to find them at all.

After spending over two days on this issue, and seeing that other developers were raising the same issues we have experienced, while nobody managed to find a working solution, we have decided to disable the certificate verification. Interestingly, this was also the only workaround the community managed to find[16, 18, 19].

Afterwards, everything started working, and we started configuring the HPA itself. The setup was easy, as the HPA configuration only needed information about the resource limits, for when it should start creating new pod instances, information about which applications it should monitor, and maximum and minimum number of those application instances that should exist. Afterall, we still want at least two instances running even when there is a low traffic in order to keep the availability high, and in case of performance errors, we do not want to overload our servers because HPA created too many pod instances, so we limited the maximum number of pods that can run at once.

The last step was to also create hardware resource limits on applications that we want to automatically scale. This means setting maximum resource values which no pod should exceed. For example, the website applications should never use more than two CPUs. This allows the HPA to then understand that if an application is using close to the maximum resource value, we need to create another instance.

Currently, we have not been in a situation where we would use the HPA. Most of the traffic requests are from the developers, web crawlers or botnets. We do have lots of traffic from the video feed, but this traffic does not considerably influence the CPU or memory usage, so the HPA does not activate. We were however successful with testing the scaling. We used an HTTP load generator *Hey*[24] to create artificial traffic, and we could observe the HPA increasing the number of pod instances that would then help with processing the traffic.

Overall though, we believe that integrating HPA into our cluster saves developers from future worries about the performance of the platform. We have not had any problems with it so far, but that does not mean we cannot unexpectedly experience it. Now we are however prepared for it, and able to manage larger amounts of traffic even in case the developers will not be available.

Additionally, the HPA is a fairly crucial part of the whole Kubernetes autoscaling

[24]https://github.com/rakyll/hey

feature. Without it, the applications would not automatically adapt, and automatic scaling of VM would never be executed, unless the developers would manually increase the number of application instances. In that case however, the developers might as well manually increase the number of VMs, thus, it would be rather impractical.

In terms of limitations, there is only one we were able to identify, and fortunately it is not as critical. Similar to the case of Kubernetes self-healing section 3.6, each new service requires its own HPA instance, which requires it to be manually configured. The newly introduced service also needs to have its resource limits in order to know when the HPA should scale it up. The configuration is not problematic, but finding out the limits is. Afterall, each application behaves differently, which means that each application will have different limits and resource requirements. We have not managed to find any concrete guidelines on how to estimate these limits, so it is up to developers to decide when a single instance of an application reaches its maximum usage. The PG stack can be very helpful to estimate these limits.

3.7.2 Cluster Autoscaler

Kubernetes is supposed to simplify management of applications running on multiple VMs. It can create a logical group of related VMs, which is referred to as a *cluster*. However, Kubernetes is not capable of increasing or decreasing the size of the cluster on its own, i.e. the number of VMs it has under its control. This is where Cluster Autoscaler (CA) becomes helpful.

There are many cloud providers capable of hosting Kubernetes, and anyone can create their own infrastructure hosting service using OpenStack[25]. However, each cloud provider usually has very different APIs and authentication systems that control how the VMs are created, or who should be able to create them. In this context, the CA can then be thought of as aggregating all of this functionality for each cloud provider together.

The CA is responsible for holding all the logic for how to create and destroy VMs for each cloud provider, and how to properly authenticate with each one of them. Thus, it allows automating the process of creating and destroying VMs. Additionally, the CA is capable of communicating with the Kubernetes cluster it is running in, which it uses to monitor the pods that run on that cluster. When the CA notices there are some pods that cannot be started because there are no free resources, such as CPU and memory, the CA sends a request to the cloud provider to create a new VM. Similarly, the CA is able to deduce when there are too many VMs running in the cluster, for example when the average CPU usage of all the nodes is below 50%, and if viable, it will shut down one of the VMs in the cluster.

A very important thing to note though is that the CA is not capable of registering new VMs into the cluster, or deregistering them from it. It is fully up to the developers to find ways of automatically joining the newly created VMs into the cluster, and the same goes for removing shut down VMs.

[25]https://www.openstack.org/

We have described what HPA is in our previous subsection 3.7.1, and CA is meant to work in tandem with HPA. Although arguably, they are both able to work alone, it is usually impractical to implement one, but not the other. Ultimately, if you are not scaling your applications with HPA, there are rarely cases when developers scale the applications manually, and then let the CA take care of adjusting the number of VMs.

Likewise, if your applications are scaled dynamically using the HPA, they have only a limited flexibility in terms of resources they can consume at any point in time. If they run out of memory, they cannot scale anymore, but with CA the limit is increased, and can be stretched basically until you hit the hardware limits of your cloud provider.

Scaling of the VMs then helps you adjust the number of VMs you cluster. This can be very helpful in situations where you suddenly have to add more processing power. As a developer you can experience unexpected events, or even periodical events, which increase traffic on your website. For example, when people have to submit some documents at the end of the month, the traffic is usually higher than at the beginning of the month. This automated scaling ability can then free the developers from manual VM instance adjustments, and save their time, while also saving VM renting costs.

Another thing to consider is that with the implementation of the CA, the developers do not have to manually operate and adjust the Terraform files. They are not necessarily difficult to configure, but managing them, especially with larger configurations, can easily lead to misconfiguration problems. Naturally, any mistakes cost us even more time, and not all developers have experience with Terraform, as it is used mostly when the cluster is being initiated. So overall, the chance of making errors when automatically scaling are lowered, and developers do not have to spend time learning Terraform.

Setup of the CA is also required only once. There are situations where you would need to adjust it, but the majority of common actions, such as updating applications or creating new ones, do not influence the scaling workflow. Therefore, any adjustments are rare, and our cluster can handle performance fluctuations for prolonged periods of time. Overall, the automated scaling might not bring as many direct advantages as the PG stack or EFK stack did, but it is much more helpful in preventing further mistakes over the long term.

A well known feature, and an advantage of cloud providers, is that they are able to offer seemingly limitless amounts of hardware. Therefore, any applications you might be running, are much easier to scale than with self-managed hardware solutions, and Kubernetes is capable of managing these cloud deployments. Unfortunately, in our case allowing our cluster to be dynamically scaled based on our needs was the most complex solution we had to implement.

According to the documentation of CA, we should be able to allow the automated deployment of new VMs with just a few steps. We should adjust a few authentication parameters, then an application capable of monitoring pods in our cluster should be able to deploy, and automatically manage the number of VMs in our cluster.

Unfortunately, the documentation fails to mention quite a few crucial details, although it might be a consequence of the high number of unique cloud providers[20]. Afterall, each major cloud provider seems to have their own unique API format for allowing creation of new virtual machines. Thus, each cloud provider needs its own documentation, and a CA module responsible for correct communication with the APIs. Both of which are usually maintained by developers of the related cloud provider that the company chooses.

After we have specified our authentication details into the CA configuration, we had to create Virtual Machine Scale Sets(VMSS). The VMSS is a concept unique to Azure cloud, but only in format, as other providers have similar concepts. The core idea of VMSS is that it represents a homogenous group for VMs. In other words, all of the VMs that are created using the VMSS are fully identical, and therefore operations executed on top of them are able to assume some parameters. The CA has multiple configurations that function, but using the VMSS deployment, we managed to decrease its overall complexity.

This complexity was however partially transferred into the configuration of the VMSS. As there are quite a few things that need to be configured in the VMSS, we will be describing only the most crucial ones. Those are the network, tags, the deployment script, and the naming scheme.

First, the network of the VMSS is required to be identical to the one of our main cluster. In this case the main cluster represents the initial three VMs that we deploy every time we want to have a working platform. In other words, there are always at least three VMs that are running, and they represent the core of our platform which is running at all times.

Otherwise, there would be simply no hardware to run our applications on, and the VMSS is an addition to these three VMs. The VMSS has to be within the same LAN, as otherwise the Kubernetes would not be able to deploy properly. At the moment we are not sure why this is the case, but we believe that it is a limitation of the Kubespray[26] deployment tool that we use to set up and scale the Kubernetes cluster. If the VMSS and the main cluster are not on the same LAN, the Kubernetes configuration is misconfigured, and the applications are not able to start.

Second, after deploying the CA, it tries to automatically discover any VMSS that it is supposed to monitor. This is ensured by specifying a VMSS name tag that the CA should look for. Therefore, the VMSS that you want the CA to manage need to have the same name tag you have given to the CA.

Furthermore, the VMSS needs tags that specify the maximum and minimum number of VMs it can manage at once. These options ensure that in case of errors, the VMSS will not scale over the maximum allowed number of VMs, and we will not have to pay astronomical bills because the CA has unknowingly created hundreds of VMs. Similarly, we want to keep a few VMs running at all times, for example if we had decided that our main cluster should be larger, but we do not want to redeploy it. Unfortunately, the details specifying what the tags should be on the VMSS are completely absent from the documentation[20].

Third, the deployment script used for VMs created by VMSS. After properly configuring the VMSS and the CA with regards to the network and the tags, we are able to automatically create and destroy VMs. The workflow is that after the CA notices that there are some pods that cannot be fully deployed, because there are not enough hard-

[26]https://github.com/kubernetes-sigs/kubespray

ware resources, it sends a request to the VMSS object on Azure, and requests a new VM to be deployed. After the VM is created, the CA reports it as visible, however, at this point, CA nor we can actually use it. The one detail left out by the CA documentation, and actually the detail for which we have ourselves created a contribution in the CA source code repository, is that it is our responsibility to also register the newly created VM into our cluster[21].

Therefore, we had to somehow automatically register the newly created VMs into our cluster. This was not a trivial task when done manually, and executing it in an automated fashion with focus on robustness, is another added challenge. The VMSS does provide the ability to specify a script that should be executed when a new VM is created, and it is what we have used in the end, but there are many challenges we had to solve. The technology which allows us to execute scripts when a VM is started is called *cloud-init*[27].

We will only mention the problems with *cloud-init* because explaining all of the problems we had encountered would be too extensive. The problems we have encountered are an incompatible format, that was for us previously unknown, and that requires us to use YAML in order to specify *bash* scripts, and the escaping rules for the bash script we had to define in the unknown format. Additionally, we needed to know when this script is executed in terms of the VM's booting process. If it was too early, our Kubernetes deployment might fail, and if it is happening every time a VM boots, we also have to provide a functionality that avoids running it twice.

We will describe the overall process of the deployment later, but the last part of VMSS configuration has to be described before. As we have mentioned, the VMSS represents a group of VMs that are identical in terms of hardware and configuration. This allows the VMSS to automatically create new VMs without worries about inconsistencies that would prevent the VMs from working identically. In other words, the VMSS aims to create VMs that are able to execute applications, whose execution results will be as deterministic as possible. This capability however does not work in our case.

Specifically, one of the areas where the VMSS tries to keep the VMs similar is their naming scheme. As the VMSS creates new VMs, they all have to follow a certain naming scheme, and in our case it was the VMSS name prefix, followed by a *Base36* ID[28]. Unfortunately, the *Base36* ID parts of the VM names that the VMSS generated, were not always in a format that Kubernetes was able to process. The consequence was that after our VMSS has created 10 VMs, the 11th and further VMs were not able to join our cluster. Of course this was unacceptable, and we had to fix it ourselves within the *cloud-init* script, as the infrastructure that Azure automatically created for us was unusable.

Overall, the final *cloud-init* script which we had to implement contained a lot of functionality. We had to first determine whether the hostname of the VM that was created was valid, if it were not, we generated a valid one using hashed date and time. After we have restarted the VM, in order to apply the hostname change, we had to extract

[27]https://cloud-init.io/
[28]https://en.wikipedia.org/wiki/Base36

details from the VM that we would need in order to register it using Kubespray. Finally, we would send an HTTP request to our service that would take care of registering this newly created VM into our cluster. As an example of the deployment script's complexity, below we had included a small code snippet of it:

```
runcmd:
 - |
   cat > /etc/rc.local <<'EOF'
   #!/bin/bash
   MASTER_NODE_IP="scaling.beelivingsensor.eu"
   LOCALIP=$(ifconfig eth0 | grep -Eo 'inet (addr:)?([0-9]*\.)
      {3}[0-9]*' | grep -Eo '([0-9]*\.){3}[0-9]*')
   curl -kLd "{\"localip\":\"$LOCALIP\", \"hostname\":\"$HOSTNAME
      \"}" -H "Content-Type: application/json" -X POST "http://
      $MASTER_NODE_IP:60000/scaleUp"
   EOF
```

At this point, in implementation of automated scaling of our cluster, we have managed to create a VMSS that is able to create almost identical VMs. The VMSS is controlled by a CA, which we have successfully deployed, and we have finally managed to ensure that our newly created VMs are able to join our main cluster. The final step is to actually join the VMs into the cluster, so that the CA is able to consider them as registered. Otherwise, we would not be able to deploy our applications on them, and the CA would delete them after sometime because it would not see the VMs through the Kubernetes API.

As we have already mentioned, the VMSS deployment script collects relevant information, and sends it to one of our applications running on our main cluster. This application was created solely for the purpose of implementing the autoscaling Kubernetes feature. Similarly to the VMSS deployment script, the exact functionality of this script is too complex to fully explain in this thesis.

However, on the higher level, the application is able to hold the current state of our whole cluster, which includes the main cluster and the VMSS VMs that have been registered so far. The application then actively listens for any requests from the newly created VMs, which would signal that we need to integrate them into our cluster. After the application obtains such a signal, it adjusts the state it holds, and uses Kubespray to properly configure the newly created VM, and adds it into our cluster so that we can deploy new pods onto it.

Our service is supposed to handle deregistering of VMs from our Kubernetes cluster as well. This case is however somewhat complex. We are not explicitly removing VMs from our cluster, instead we overwrite the VMs that no longer exist with the newly created ones.

Usually, when removing nodes, we have to also start a removal process, but in our case the CA deletes the VMs without any warning. So far we were not able to find a way on how to automatically trigger the removal process, and even if we did, there are a number of other problems that we would have to solve, as by the time we start the removal process, the VM could no longer exist.

Therefore, we do not remove VMs from our cluster, and instead overwrite them if new VMs are being added. In case the CA deletes a VM, all of the pods are simply marked as *terminated*, and they are moved onto the healthy VMs. Overtime, these missing VMs will be overwritten by the existing ones. During testing, we did not find any issues with this setup, which leads us to believe that it is stable and functional.

As you can imagine, the overall process contains a lot of functionality. Implementation, and especially testing was very time consuming. We have also encountered quite a few bugs in the Kubespray which were crucial for a successful execution of our automated scaling system. As such, the amount of work we had to put in was considerable.

However, with the autoscaling completed, there is theoretically no longer a limit to how many users our service can manage. Unfortunately, we were not yet able to experience the automated scaling due to lack of natural traffic. We were able to trigger the automated scaling by generating artificial traffic, but we have yet to reach natural traffic levels that would trigger it.

At this point however, the platform will be able to run autonomously with very minimal maintenance efforts from the side of developers. Additionally, we believe that the auto scaling feature is simply too crucial, and it would have been implemented in the future in any case. Thus, we have saved time for future developers, especially if they would not be experienced with Kubernetes.

There are many moving parts, which only increases the complexity of our cluster, and with such complexity, we also have to mention a few limitations. As we have mentioned, the CA documentation is rather sparse, and omitting important details. We have tried our best to describe these details, and the whole implementation in our documentation, but we are not sure how long it will stay relevant. Kubernetes is an ever evolving project, and with that the CA evolves as well.

At the moment everything is operational, but considering how complex the functionality is, we are not able to estimate if our solution will stop working with the introduction of new versions. Therefore, it might be much harder to keep our platform updated.

Currently, we are able to scale, but our process unfortunately does not work well when trying to add multiple new VMs at the same time. The Kubespray is capable of registering multiple VMs at once, but our registration service has problems with adding new VMs while the registration process is running. Therefore, we are not able to add more than one VM every 20 to 30 minutes. However, we believe that in case when we would need more than one new VM every 30 minutes, the developers would be already monitoring the platform manually because it would be an unusual occasion.

Initially, we were also planning to create an automated scaling system for VMs that have access to GPU cards that we can use for processing our machine learning models. At the time of writing however, these machines were not fully accessible, and we were not able to test whether our process works with them as well. The auto scaling process should not be different, except the VMSS should use a different type of VMs, but since we were not able to test them, we do not know whether it works.

Lastly, we were not able to fully automate the deployment of the CA. Large parts of our platform deployment process are automated, but the deployment of CA, and creation of VMSS needs a manual setup every time we create a new cluster. We believe that it is possible to fully automate the process, and large parts of it are already automated. However, certain steps are difficult to automate, and if they were misconfigured, they could cause a lot of problems. Therefore, combined with the rare need to create a new cluster, we have not fully automated the deployment process of the cluster autoscaling.

3.8 Continuous Delivery and Deployment

As part of our efforts to help developers automate common tasks, and therefore decrease their maintenance effort as per RQ 1.4., we have identified the *continuous deployment and delivery* (CD[29]) as one of such areas. The developers are constantly creating new updates for the applications running on our platform. After the updates are properly tested, and the developers want to deploy their new versions into the production environment where real users can use them, they also need to execute two other steps.

First, we need to create Docker images, which represent the application. The Docker images contain all the necessary libraries and executables that are needed for a successful functioning of the main application. The images are built by creating a *Dockerfile* that contains instructions on how to build an image, and then we just need to instruct Docker to build the image.

At this point though, the newly created Docker image can be used only locally. In order to use it in our cluster, we have to also upload the image into a public Docker repository such as Docker Hub[30]. The process of building and uploading images is called *Continuous Delivery.* The goal is to have the application ready for deployment, but not deploy it[39].

Second, after the images are publicly accessible, we need to also notify the Kubernetes about the new version being present, and also instruct Kubernetes to deploy it. Any further changes are done automatically by Kubernetes, the developers only have to make sure that the deployment was successful. This process is called *Continuous Deployment*, and in combination with continuous delivery, it allows developers to write updates for their application, and then easily integrate them into a live environment[49].

Therefore, this process is rather common and frequently used. There are not many things that can fail in this process, and thus, automating it would free developers from this repetitive work. Additionally, it would also decrease the risk of any human caused errors, such as typos or accidental deletion of applications. Automating the CD process can also free the developers from the requirement of connecting into the cluster itself, which can help us keep the cluster secure, as we will decrease the number of people with access to administrative commands.

Overall, the automation process is not difficult to implement, which is also one of the main reasons why we have decided to implement. This process is simplified especially by the infrastructure that GitLab[31] provides. They allow us to specify when our code

[29]Usually, both continuous deployment and continuous delivery are implemented simultaneously, which is why their initials "CD" are used to refer to their combination, rather than their individual parts.
[30]https://hub.docker.com/
[31]https://docs.gitlab.com/ee/ci/

should be run through the CI/CD pipeline, and what commands should be executed in each stage.

We have already configured the testing stage, as part of our previous work on the Beelivingsensor platform, in which we execute tests to see whether our code does not include any bugs. Our research however adds new *delivery* and *deployment* stages, and naturally, these stages refer to the continuous delivery and deployment concepts respectively.

As we have mentioned, the part of the delivery stage is building of Docker images and their upload. Therefore, we have automated the process of building them, as we already have the building specifications for each application. The only difference from our usual manual process was that we had to label the Docker images with the latest *git* commit hash, in order to somehow denote what version of the Docker image was built. Lastly, we had to set up authentication for our shared Docker Hub account, and then upload the images.

The *deployment* stage was a bit more intricate. The difficult part was authenticating with our Kubernetes cluster. We had to create an administrator account on Kubernetes, and then properly configure the connection between GitLab and our cluster. Afterwards, we also had to create a custom environment for the *deployment* stage because the default environment did not include functionality required to communicate with our cluster.

Finally, we specified which applications we would like to keep regularly updated, and we created a logic that verifies that our applications were deployed successfully, and we are able to access them. If any stage fails to execute successfully, the whole CD process is stopped, and we are notified about the failure.

Shortly after implementation we have started using the CD process frequently. However, after a few days we have decided to disable it. The CD process is still present, and still functional, but we have decided that it would be better if we manually trigger the whole process.

This is a bit counter intuitive, as our initial goal was to fully automated the CD process, so that developers do not have to manually execute anything. This benefit is still there to a certain degree. The only steps the developers have to take is to initiate the process. The decision to disable the CD was a result of our source code committing practices.

We were not expecting to make this decision, but it turned out that we push multiple commits to our main repository, although the commits are not expected to work. Previously this was not an issue, as only our tests were executed after we had committed new code. With the introduction of the CD process however, we have automatically started building images, uploading them and deploying as well. This was not necessarily an issue either, but the CD stages were quite lengthy, especially the building process, and we were trying to minimize the time we have spent executing commands.

The CI/CD GitLab functionality is free of charge and available for any registered users, but each repository has a limited number of execution time. In our case, we usually ran out of our free execution time, and in order to continue using it, we had to actually pay for it. Therefore, we have tried not to waste it, and since the CD process was so time consuming, and yet we did not always need our applications to be deployed right away, we have decided to disable the CD process by default.

In terms of limitations, the only issue is that whenever we create a new Kubernetes cluster we need to reconfigure the GitLab authentication. With the new cluster we obtain new authentication certificates, and the previously used credentials from GitLab would not be accepted anymore. Since the authentication configuration is done through a web UI, we were not able to find a way to automate this process yet. Considering that we rarely create new clusters, we also believe that the disadvantages are not as detrimental.

4

Evaluation

The improvements that we have introduced as part of this thesis are all described in detail in their respective sections. In order to show, however, that our improvements are functional, and fulfilling the goals we have set for ourselves, this chapter explains two experiments that we have created. The two sections also mention any unusual observations, and results of those experiments, in addition to what we aimed to achieve by conducting them.

4.1 Chaos Engineering

Our work aims to improve the maintainability, and robustness of the Beelivingsensor platform. We have provided concrete solutions in chapter 3, however, we would like to also explicitly show that they are helpful. Expected benefits of each solution are of course listed alongside their descriptions, and we have also described how useful they were in our own cases, but we would like to provide a more replicable measure of success. Cases where we found the tools helpful are in our opinion common, but for the sake of leading a scientific research, we would like to provide more explicit cases that can be replicated with little effort.

In this section, we will show that our platform will stay operational even in the case of errors within our applications, or termination of the pods from the Kubernetes side. Moreover, Kubernetes will be able to recover from these problems, and restore the functionality automatically, while also notifying the developers about long lasting outages or abnormal events.

Afterall, if an application crashes or if traffic spikes occur, and they are quickly and automatically resolved, there is no need to alert the developers. However, in cases where an application is unavailable repeatedly, we need human intervention to investigate the issue. Lastly, we will also test that we are able to browse metrics in Grafana during these simulations, to show that even with an ongoing outage, we are able to debug and understand what is happening.

To properly showcase this functionality we use the Litmus[1] to simulate errors. Litmus is a tool deployed onto Kubernetes, which is capable of simulating various errors in the

[1]https://github.com/litmuschaos/litmus

Kubernetes, and the applications that run on it[24]. There are multiple events, such as creating artificial network problems, hardware failures, and application failures that Litmus can simulate. We have used it to simulate pod termination, application failure, increased CPU usage, and VM crash.

The success of the failure experiments is determined by the Litmus. Each experiment that simulates an error defines when the applications recovered successfully. For example, in the case of pod termination, if there is the same amount of pods successfully running at the end of the experiment, as at the beginning of it, the Litmus marks it as a success[25].

Additionally, during each experiment, we are monitoring the state of Kubernetes manually using terminal tools and Grafana, and we run the *Hey*[2] tool in the background. The *Hey* then keeps constantly requesting our website, to make sure that it is reachable during each experiment, and in the end we are able to see the statistics for the requests. For each experiment you can see a related output of the *Hey*. Moreover, you can see how the Litmus communicates with other components in Figure 4.1.

Pod Termination. The pod termination test simulates any events where Kubernetes might request any application to terminate. This is different from the application failure because it is the Kubernetes that requests the application to terminate, and the request allows the application to finish its operations. This is often referred to as *graceful termination*.

Therefore, it is up to the application to terminate when possible. The major benefit of graceful termination is that it can prevent interrupting applications which are in their critical sections. Those are processes, such as writing to a file or communicating with a database, and therefore graceful termination can prevent inconsistent states and data loss among other problems.

A slight inconvenience is that the Kubernetes expects the application to eventually terminate, and if an application for whatever reason does not terminate by itself, it will be left alive. In other words, the trust is placed upon the application, and that it will finish as soon as possible, but if that does not happen, it is possible that the applications keep consuming precious resources, and they can block further Kubernetes processes.

We have configured the pod termination experiment to run for 30 minutes, and requested it to terminate our pods every 30 seconds. We have focused on our web, model analysis, and FTP server applications. During this time, pods of each application were randomly terminated, and Kubernetes then created new pods in order to keep the applications operational.

The results of *Hey* can be seen in Figure 4.2. During the experiment we did not notice any problems with connectivity, as the incoming user requests were redirected only towards the pods that were ready to accept traffic. In the *Hey* results you can see 154, 502 responses that indicate errors in the application. These errors indicate that the traffic was routed to pods that were just starting, meaning the pod was healthy and running, but the web did not fully start yet[26].

This is an issue on the application level, and not the platform. Meaning that we would have to improve the application to fix this error. We however believe that it does not

[2]https://github.com/rakyll/hey

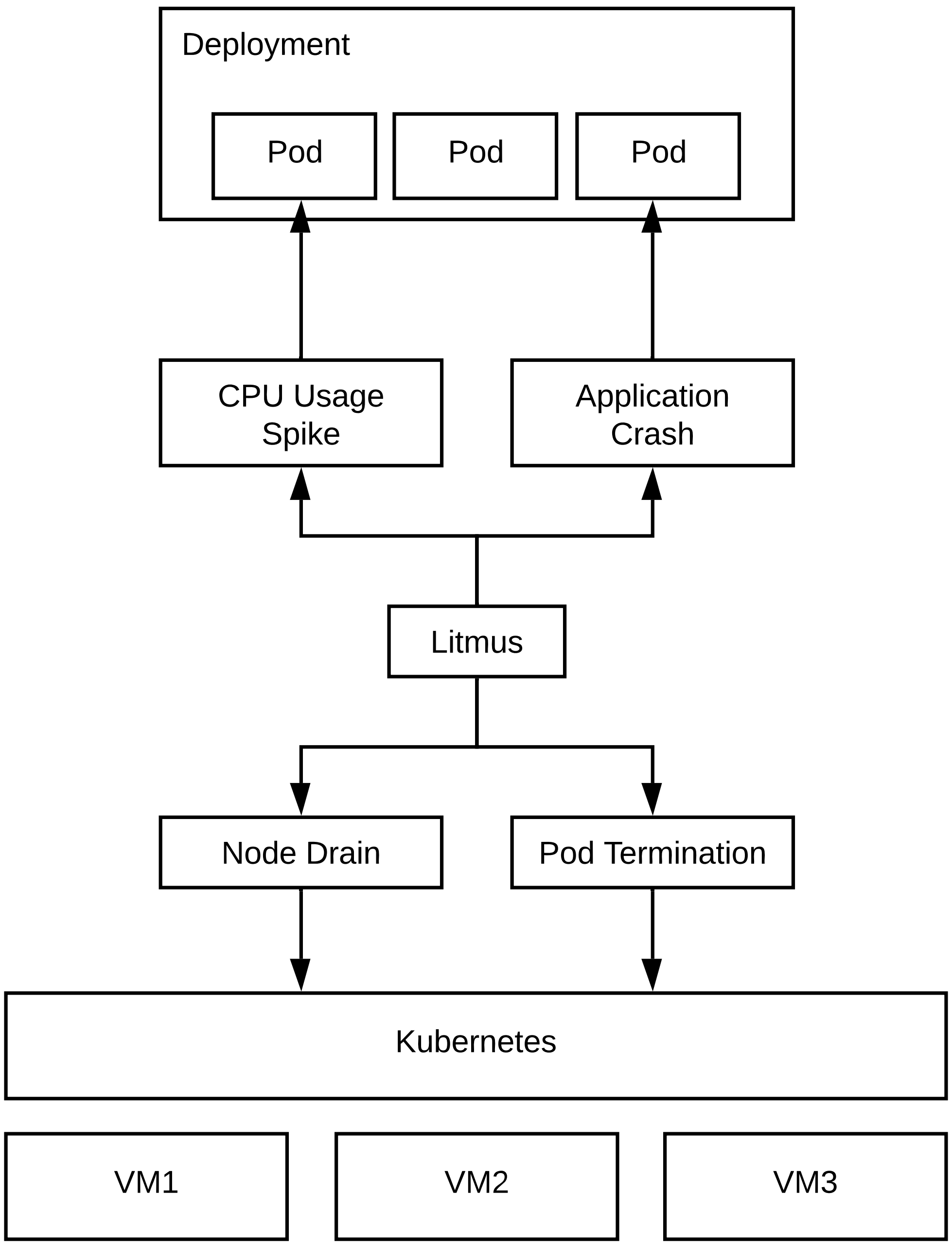
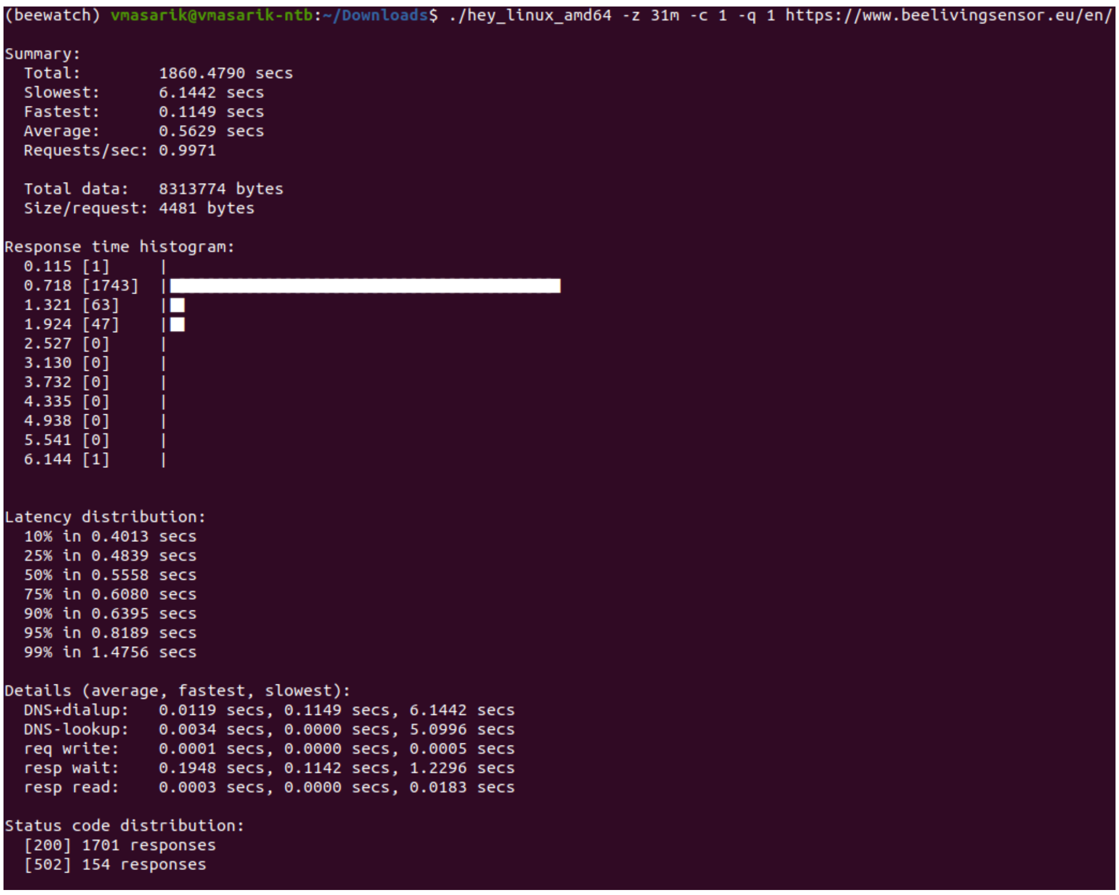


Figure 4.1: Positioning of Litmus in the context of Kubernetes, and other applications.



```
(beewatch) vmasarik@vmasarik-ntb:~/Downloads$ ./hey_linux_amd64 -z 31m -c 1 -q 1 https://www.beelivingsensor.eu/en/

Summary:
  Total:        1860.4790 secs
  Slowest:      6.1442 secs
  Fastest:      0.1149 secs
  Average:      0.5629 secs
  Requests/sec: 0.9971

  Total data:   8313774 bytes
  Size/request: 4481 bytes

Response time histogram:
  0.115 [1]     |
  0.718 [1743]  |■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
  1.321 [63]    |■
  1.924 [47]    |■
  2.527 [0]     |
  3.130 [0]     |
  3.732 [0]     |
  4.335 [0]     |
  4.938 [0]     |
  5.541 [0]     |
  6.144 [1]     |


Latency distribution:
  10% in 0.4013 secs
  25% in 0.4839 secs
  50% in 0.5558 secs
  75% in 0.6080 secs
  90% in 0.6395 secs
  95% in 0.8189 secs
  99% in 1.4756 secs

Details (average, fastest, slowest):
  DNS+dialup:   0.0119 secs, 0.1149 secs, 6.1442 secs
  DNS-lookup:   0.0034 secs, 0.0000 secs, 5.0996 secs
  req write:    0.0001 secs, 0.0000 secs, 0.0005 secs
  resp wait:    0.1948 secs, 0.1142 secs, 1.2296 secs
  resp read:    0.0003 secs, 0.0000 secs, 0.0183 secs

Status code distribution:
  [200] 1701 responses
  [502] 154 responses
```

Figure 4.2: Hey output for Pod Termination experiment.

endanger the long term maintainability of the platform. Continuing the assessment, our manual observations did not find any errors either, and we were able to monitor the platform at all times using Grafana. Lastly, the results of the experiment by the Litmus reported success as well.

Application Failure. This experiment aims to simulate unexpected crashes of applications that we run. Such cases can happen if the applications contain bugs, and as part of this thesis, we wanted to make sure that even in case of such bugs, the platform will be functional anyway. Therefore, we hope to show that the maintainability of our platform is improved because our platform will be able to tolerate errors, and also recover from them by restarting the applications.

The experiment is similar to pod terminations, except, as already stated, we will be killing the processes without any warning. Meaning, the applications will be terminated completely at random, and there is a possibility that we will cause inconsistent states or data loss. With that in mind, we believe that our web application is capable of withstanding such situations, while we would like to avoid creating unnecessary failures in our other applications. Therefore, we will cause failures only in the web application.

The configuration of the experiment is almost identical to the pod termination. We will be crashing our web application for 10 minutes, every 30 seconds. The two differences from the pod termination is that the crashes will not wait for the application to end by themselves, and that we are going to be crashing the applications only for 10 minutes. The reason for running the experiment only for 10 minutes is that the application crashes cause Kubernetes to delay their restart[29].

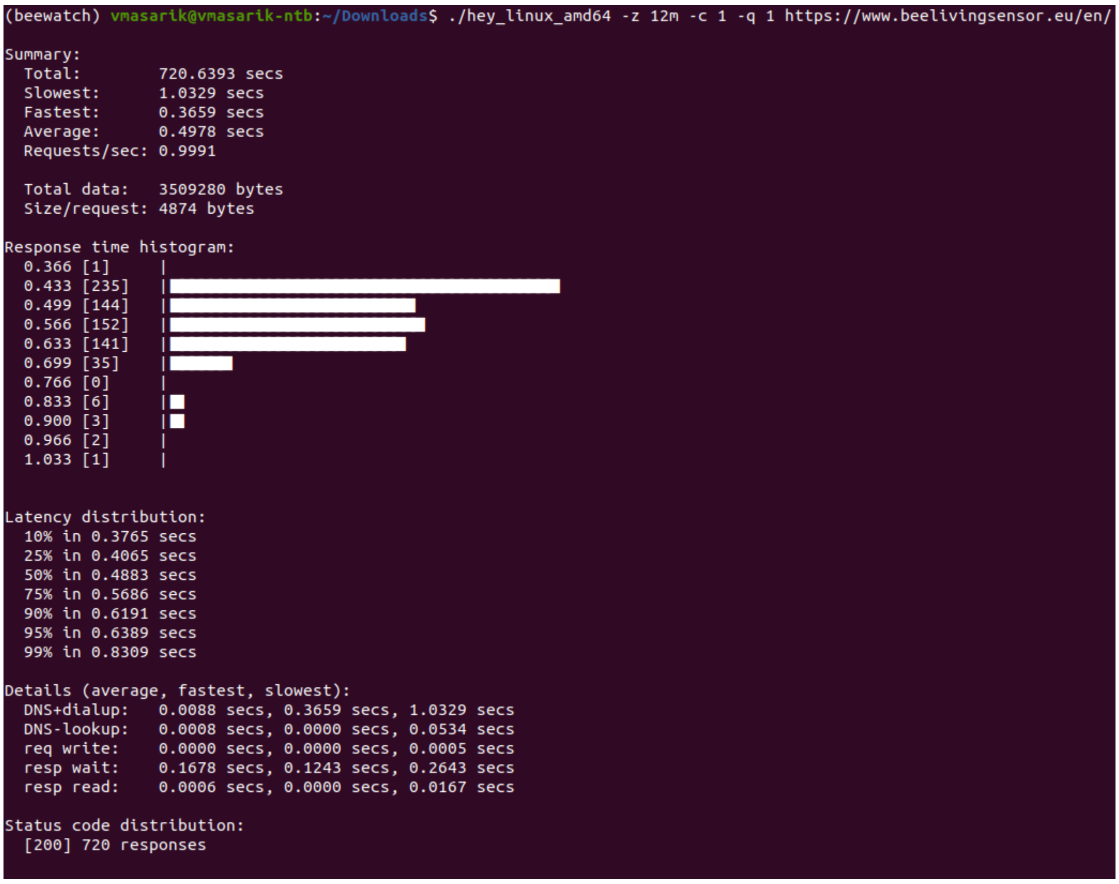
This delay slowly increases as well, and therefore, if we keep crashing the applications constantly, it quickly reaches a few minutes long restart delay. The reason for increasing the restart time is that if an application keeps failing soon after starting, restarting it does not change nor help the situation. This creates a problem that if the Litmus does not see the application running within a certain time, it will conclude that the application is not healthy, and it concludes that the experiment failed.

That is not the case however, as after some time the application again restarts, and continues to function successfully. We have tried to find an option in Litmus that would increase this waiting time, but no matter what we changed, the experiment continued as expected until around 12 minutes, when the application was restarted for the 7th time. After this 7th time, the Litmus always timed out waiting for the application to restart, and considered the experiment failure. Therefore, we have limited the experiment to 10 minutes only.

As we have mentioned, the Litmus results reported success for the experiment, as everything was working afterwards. During our observations, we have not noticed any errors, or unusual patterns. The output of *Hey* can be seen in Figure 4.3, and as you can see, all of the requests were successful, and we had no errors, nor large delays.

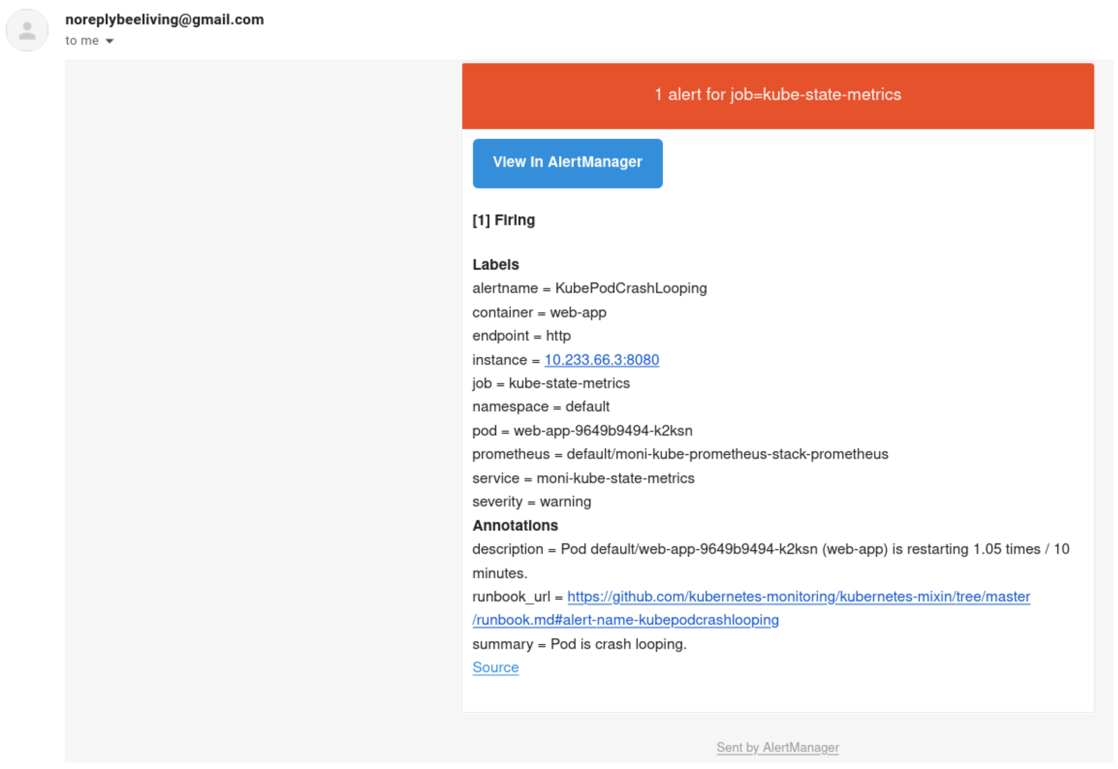
This experiment is special from the others because if any application keeps failing for more than 10 minutes, which is what we exactly forced them to do, we will receive notifications about these failures. Naturally, we have started receiving them as well. You can see the content of the email notifications in Figure 4.4.

Node Drain. The node drain experiment aims to simulate a sudden shut down of



```
(beewatch) vmasarik@vmasarik-ntb:~/Downloads$ ./hey_linux_amd64 -z 12m -c 1 -q 1 https://www.beelivingsensor.eu/en/

Summary:
  Total:        720.6393 secs
  Slowest:      1.0329 secs
  Fastest:      0.3659 secs
  Average:      0.4978 secs
  Requests/sec: 0.9991

  Total data:   3509280 bytes
  Size/request: 4874 bytes

Response time histogram:
  0.366 [1]     |
  0.433 [235]   |■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
  0.499 [144]   |■■■■■■■■■■■■■■■■■■■■■■■■■
  0.566 [152]   |■■■■■■■■■■■■■■■■■■■■■■■■■■
  0.633 [141]   |■■■■■■■■■■■■■■■■■■■■■■■■
  0.699 [35]    |■■■■■■
  0.766 [0]     |
  0.833 [6]     |■
  0.900 [3]     |■
  0.966 [2]     |
  1.033 [1]     |


Latency distribution:
  10% in 0.3765 secs
  25% in 0.4065 secs
  50% in 0.4883 secs
  75% in 0.5686 secs
  90% in 0.6191 secs
  95% in 0.6389 secs
  99% in 0.8309 secs

Details (average, fastest, slowest):
  DNS+dialup:   0.0088 secs, 0.3659 secs, 1.0329 secs
  DNS-lookup:   0.0008 secs, 0.0000 secs, 0.0534 secs
  req write:    0.0000 secs, 0.0000 secs, 0.0005 secs
  resp wait:    0.1678 secs, 0.1243 secs, 0.2643 secs
  resp read:    0.0006 secs, 0.0000 secs, 0.0167 secs

Status code distribution:
  [200] 720 responses
```

Figure 4.3: Hey output for Application Failure experiment.


noreplybeeliving@gmail.com
to me

1 alert for job=kube-state-metrics

[1] Firing

Labels
alertname = KubePodCrashLooping
container = web-app
endpoint = http
instance = 10.233.66.3:8080
job = kube-state-metrics
namespace = default
pod = web-app-9649b9494-k2ksn
prometheus = default/moni-kube-prometheus-stack-prometheus
service = moni-kube-state-metrics
severity = warning
Annotations
description = Pod default/web-app-9649b9494-k2ksn (web-app) is restarting 1.05 times / 10 minutes.
runbook_url = https://github.com/kubernetes-monitoring/kubernetes-mixin/tree/master/runbook.md#alert-name-kubepodcrashlooping
summary = Pod is crash looping.
Source

Sent by AlertManager

Figure 4.4: Content of the email notification triggered during the Application Failure experiment.

a VM. In this context *node* is a synonym for VM. The experiment is not an identical simulation of a VM shut down, as the applications running on the VM are terminated by the Kubernetes, rather than an unexpected application crash as is the case of application failure experiment.

The reason being that the Litmus does not provide us with an experiment which would shut down the VM instance in reality. They do provide such functionality for two specific cloud providers, but not for Azure[28]. Moreover, the only difference between shutting down a VM instance in reality, compared to our simulation, is that the pods are terminated in a different fashion, and the VM is able to answer health requests. Therefore, if our applications are able to properly recover after the application failure experiment, which they are, there should be no problems in case of a real VM crash.

As to what happens during the experiment, the Litmus filters a list of pods that are running on the targeted node. Then it requests Kubernetes to terminate all the applications on the VM, whether our web or our infrastructure support. Additionally, the VM is marked as not fit for running new pods, therefore, all the pods will be executed on different VMs. Once there are no pods running on the VM, the experiment is finished.

The configuration specified for this experiment only requires definition of the node that should be drained, and on which VM should the experiment executing application run. Afterall, we do not want the experiment application to be terminated while on the drained node, as this would stop the experiment. In case of the node drain experiment we did not collect *Hey* output. The experiment is executed only once, and not continually. Therefore collecting this data is rather redundant as the application pods will be terminated only once, and a much better benchmark of whether our applications are available at all times are the application failure and pod termination experiments.

We have, however, continued to observe the process manually, and through Grafana. We did not observe any unexpected errors, nor alerts. Some applications, such as our network plugin Flannel and Prometheus, responsible for keeping the infrastructure operational reported timeout errors, but that was rather expected as the node drain experiment terminated their key pods.

In summary, the pods were successfully deleted, immediately recreated on the other functional nodes, the experiment was then considered to be finished, and the node was marked operational again. Afterwards, applications which were required to run on the drained node subsequently returned, and continued without any visible problems. Finally, we have listed the Litmus experiment results, and it reported success as well.

CPU Usage. The last experiment we ran was CPU resource usage. In other words, the experiment increased the CPU usage of the pods. This also triggered the automated pod autoscaler, which created new pods to cope with the increase in CPU usage. The main reason why we have included this simulation was to provide a more reliable way of replicating our results for benchmarking the improvements we have implemented.

We have already discussed the usage of *Hey* in testing the automated scaling that we have implemented, but when executing such a benchmark, in some cases the results could be different. For example, if the network connection is weak, or if the hardware that our applications might be using is less performant, the results can be lower. Specifically, the slow network might not allow you to create a necessary load to trigger the scaling, and conversely the weaker hardware can cause the applications to be overloaded even in case of slightly higher traffic increase.

This experiment however, consumes the CPU regardless of the network condition, and it allows us to define the number of CPUs to use, therefore, it is much more precise in defining the benchmarking situation. We believe that the *Hey* is a better tool for benchmarking the performance of the web applications, but for the sake of replicability, we have decided to include the increased CPU experiment as well.

The configuration includes only the number of CPUs that should be used, and how long the experiment should be carried out. We ran the experiment for 30 minutes, and specified only one CPU to be used. The CPU setting did not necessarily matter however, as our applications were limited to use only half of a CPU, thus, they were using maximum resources in any case.

As with the previous experiments, you can see the output of the *Hey* in Figure 4.5. The output shows that there were no interruptions, and all user requests were successfully processed. We have observed the experiment manually, and with Grafana as well, but have not seen anything unusual. The web application was scaled appropriately in order to average out the processing load, and after the experiment ended it decreased the number of pods as expected. You can see the CPU usage rise during the experiment in Grafana in Figure 4.6.

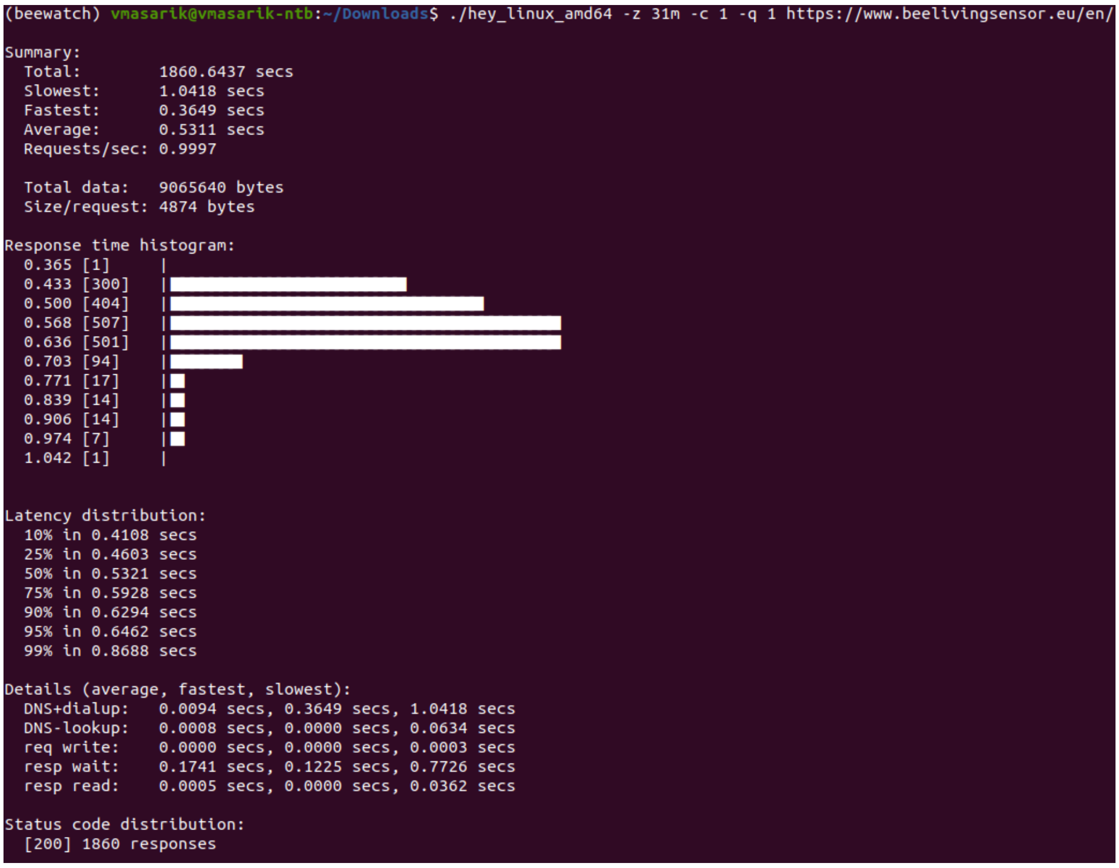
Results. As we have defined before, the goal of these four experiments was to prove that our platform is resilient enough to keep serving the user requests even when our applications keep crashing. We proved that with *Hey* outputs for each experiment, and we have described each one of them in detail.

We have also shown that Kubernetes is capable of recovering from these errors automatically. Litmus reported success for each experiment, therefore, after every test our platform reached a stable and a healthy state. Moreover, we have received email notifications in case our applications were failing for more than 10 minutes. We have monitored each experiment manually and with Grafana without spotting any unusual errors or patterns.

This shows that the tools we have implemented as part of this thesis improved the maintainability of the platform, and the need for human intervention was decreased. In our opinion these experiments are also easily replicable, and the fact that we have not encountered any errors means that everything works as intended.

4.2 Timed Debugging Experiment

As part of the thesis we have planned to conduct an experiment that would provide an objective view on the usefulness, and usability of the tools that we have implemented as part of this thesis. We understand that our new tools are not a panacea for every problem, and in some cases they even lack the functionality that the very basic terminal tools provide. However, we wanted to see whether we have improved the debugging time in the cases of common misconfiguration issues, and log search. Additionally, we wanted to see whether there are any problems that we have failed to notice, and which are easy



```
(beewatch) vmasarik@vmasarik-ntb:~/Downloads$ ./hey_linux_amd64 -z 31m -c 1 -q 1 https://www.beelivingsensor.eu/en/

Summary:
  Total:        1860.6437 secs
  Slowest:      1.0418 secs
  Fastest:      0.3649 secs
  Average:      0.5311 secs
  Requests/sec: 0.9997

  Total data:   9065640 bytes
  Size/request: 4874 bytes

Response time histogram:
  0.365 [1]     |
  0.433 [300]   |■■■■■■■■■■■■■■■■■■■■■■■■
  0.500 [404]   |■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
  0.568 [507]   |■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
  0.636 [501]   |■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
  0.703 [94]    |■■■■■■■
  0.771 [17]    |■
  0.839 [14]    |■
  0.906 [14]    |■
  0.974 [7]     |■
  1.042 [1]     |


Latency distribution:
  10% in 0.4108 secs
  25% in 0.4603 secs
  50% in 0.5321 secs
  75% in 0.5928 secs
  90% in 0.6294 secs
  95% in 0.6462 secs
  99% in 0.8688 secs

Details (average, fastest, slowest):
  DNS+dialup:   0.0094 secs, 0.3649 secs, 1.0418 secs
  DNS-lookup:   0.0008 secs, 0.0000 secs, 0.0634 secs
  req write:    0.0000 secs, 0.0000 secs, 0.0003 secs
  resp wait:    0.1741 secs, 0.1225 secs, 0.7726 secs
  resp read:    0.0005 secs, 0.0000 secs, 0.0362 secs

Status code distribution:
  [200] 1860 responses
```

Figure 4.5: Hey output for CPU Usage experiment.

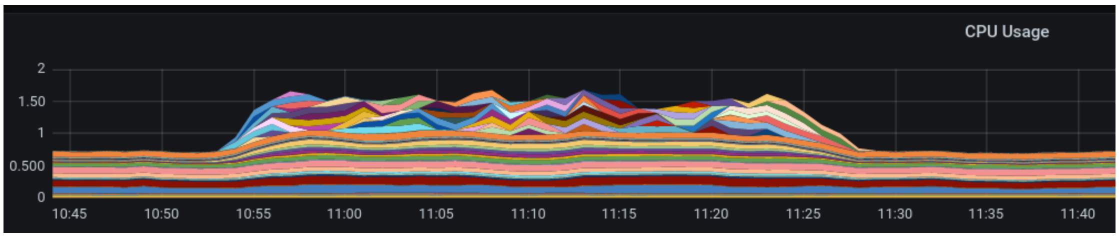


Figure 4.6: Grafana dashboard showing increased CPU usage by web application pods during CPU Usage experiment.

to change, so that the maintainability is further improved for the future developers. In other words, we have spent a long time working on our research, and we wanted to know whether there are some simple solutions which we have failed to consider, but are obvious to other developers.

Initially we hoped to just compare the time in which two developers managed to solve two debugging tasks. Meaning that we would compare the time in which they solved a debugging task using tools that come by default with Kubernetes, and any other tools that can be installed in the terminal, against the web based tools that we have implemented as part of this thesis. Specifically, the web based tools are the Kubernetes Dashboard described in section 3.3, and Kibana described in section 3.2. We will refer to the tools that come by default as the *terminal* tools, and the newly implemented tools Dashboard and Kibana as *web* tools.

The two debugging tasks that we wanted to use to compare the terminal tools with the new web tools were misconfiguration, and log search. We wanted to show that when developers use our web tools to debug such problems that they are on average faster in finding and fixing the problems. To show that we have sped up the process, we needed to first know what is the time that the developers spend on solving the issues without our web tools. Therefore, we had to have a control sample where the experiment participants would use the terminal tools, and the main time sample where the participants would use the web tools to solve the same tasks.

This would mean that we need to have at least four timed tasks for every participant, in addition to the experiment introduction, and a round of questions after each task. We expect that to properly measure and compare the timed tasks we would need around 30 minutes. We hope that the longer time span is able to negate any subjective difference between the experiment candidates, and show that the difference in time spent on debugging the problem is caused especially by the difference in tools that the participants are using.

Unfortunately, such long times would quickly culminate into each experiment lasting more than two hours. Therefore, it leads us to believe that this would cause our potential participants to be less willing to participate in our experiment. Moreover, we already had problems finding volunteers for our experiment, the reasons are explained further below in this section. Therefore, we have decided to lower the time of each task to only 10 minutes.

This decision however causes more problems in our experiment design. Ten minutes per task is a bare minimum of how long it takes to debug a problem. Additionally, with such a low amount of time per task, there are many factors that can influence the time it takes to debug a problem. With longer tasks, we would expect these factors not to be that influential, as the relative difference between the tasks would be small. With the lower time spans, if a participant has a subjective issue that causes them to spend one more minute on the problem, this results in basically a 10% difference between terminal tools and web tools. Therefore, the time difference caused by the web tools might not be as visible as with tasks that have longer duration. Rather it might represent subjective and for us irrelevant differences between the participants, such as reading and comprehension speed.

With this in mind, and in combination with other reasons described below, we have decided the time factor would be just one of the factors that we would consider in order to show that your web tools help developers. Rather, we would like to take a more qualitative approach towards our experiment, where we ask the participant questions about how they perceive our web tools in comparison with the terminal tools. This way, we would be rating the usefulness of our web tools based on participants' observations, along with how quickly they solved them. We believe that such an approach would provide us with the most objective comparison that is possible considering our situation.

Participants. Finding volunteers is not an easy task, especially if we have rather specific requirements. Specifically, we are looking for developers who are aware that Kubernetes exists, and where it might be used, but ideally they did not work with it yet. The reason for such requirements is that we want to simulate situations when future Beelivingsensor developers will not be familiar with Kuberentes, as this is the case where we want to improve the maintainability of our platform. We still believe that even more experienced Kubernetes users can speed up their debugging efforts, but it is not the core objective of this thesis.

Moreover, we are able to include only short and very basic debugging tasks in our experiment, and we expect that if the participants had some experience with Kubenretes, they would be able to guess the solutions which could make comparing the results difficult.

Additionally, during our design phase, we had a few testing trials on people with varying levels of Kubernetes knowledge, and we have observed that for participants that have never heard of Kuberentes, it was very difficult to grasp the concepts necessary for the experiment quickly enough. We had to first explain to them what Kubernetes is, and the context of where it is used, in addition to concepts like pods, services, and secrets.

They were able to understand it eventually, but initially it was too much information for them to process, and they were able to work autonomously only after they have already solved two or three experiment tasks, depending on the person. This meant that the time they took to solve a task, also included time necessary for them to understand the Kubernetes, which was something that we did not want to measure. Hence, we are looking for developers that have some knowledge of Kuberentes, but they are not able to solve our tasks too quickly. Unfortunately, it is rather difficult to find such a specific group of people.

Moreover, we want the participants to be able to use a Linux terminal, and the more experience they have, the better. As we have mentioned, one of our goals is to see whether we have missed any simple maintainability improvements. We hope that the experiment participants will show us new possibilities, and we expect that the more experience they have with using a terminal, the more likely they are to use new creative solutions for our tasks.

As previously mentioned, our initial plan was to measure the time difference between the participants who were using our web tools and the terminal tools to solve tasks. Considering the above mentioned requirements however, it became clear that finding developers willing to take part in our experiments would be difficult, especially because of the time and effort needed to conduct a somewhat useful research. Therefore, we have decided to have at least some way to evaluate our web tools, rather than conduct no experiment at all. It might not be perfect, but we still believe it will bring some value, and perhaps the future researchers would be able to avoid the mistakes that we have encountered, and they would be able to improve the experiment if necessary using our experience.

Additionally, the experiment would require the participants to have access to our clusters, which naturally endangers our security. We are able to manage a few users and their access manually, and therefore negate the danger. However, with a growing number of participants, we would need to find a different solution, which would possibly complicate the experiment even further. Thus, these considerations only further cemented our decision that we should do a qualitative study, and that we should continue with the initial plan of including only two participants in our study.

The search for volunteers was difficult as expected, and rather long. We have asked student developers on group chats, and unfortunately, we did not offer any incentives for participation. In the end, we have recruited two volunteers. Both of them were familiar with Kubernetes, and the context where it is used, thanks to where they worked, but neither of them interacted with it before. Additionally, both had at least two ears of working experience with terminal tools, although they were using it at a rather different frequency.

Experiment Design. We had a lot of relevant situations we could include as tasks in our experiment, but finding the proper tasks was harder than we initially expected. The hardest part was finding misconfiguration tasks that were identical in difficulty, therefore comparable, and tasks where the participants would be able to solve them in under 10 minutes.

Our final design included four tasks. The two logging tasks were very similar, as the participants had to use the available tools to find relevant error logs. For our first logging task, our web application was running for over a week in production. During this time we have manually triggered an exception in our web application using a well known problem, which we plan to fix in the future. This resulted in error logs, which the participants were supposed to find using available tools. To create the environment for our second logging task, we continued to use the web application pod in production, and after a few days, we have triggered again another well known error, although somewhat similar, which resulted in error logs. These exceptions were different in log output, and the time when they occurred. Therefore, both tasks were easily comparable, as they had almost identical processes of debugging, and the only differences were output text that the participants were searching for, time of occurrence, and the tools participants would use.

Concerning the two misconfiguration tasks, our initial plan was to present the participants with any misconfiguration issue, and then ask them to find the cause of the issue, and fix it. The problem was that in both cases where the participants would use web tools and terminal tools, the issue would be the same. Therefore, the second time they would be solving the same debugging task, but with a different tool, they would have been faster because they would have already known the solution. Meaning, the part of

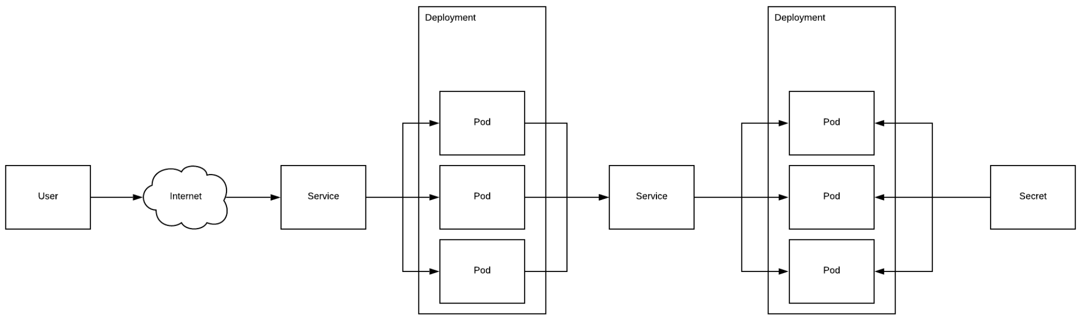


Figure 4.7: Graph provided to the experiment participants, showing the relations and communication between various Kubernetes resources.

the task which required the participants to find the problem, would be already solved if they were solving the same issues again. This forced us to find different misconfiguration cases, however most of them would be fairly incomparable.

We were fairly successful in the end, but even after multiple improvements and re-designs, we were not able to create similarly good tasks as with the logging tasks. Instead, we focused on two different types of misconfiguration issues. The process of fixing the configuration is quite similar in all cases because the participants only have to find and fix the one place where the configuration is incorrect. The less similar part is debugging the said problem. Here we have opted for misconfiguration of authentication details, and misconfiguration of connection details.

In both cases, the important Kubernetes resources which held the configuration were somehow directly related to the application. You can see the *Service* and *Secret* in Figure 4.7. For both tasks the users were supposed to first read the logs to understand the problem, then identify the resource which could be misconfigured, find the misconfigured values, and adjust them.

The difference in these two tasks is that the task related to the *Service* resource required the user to log into the container using a command, which we have provided in the task definition, and execute a command in order to read the logs. In comparison, the task, using the *Secret* resource, only required the user to print the logs that were already present in the pod. Therefore, the *Service* task would theoretically be longer

The tasks were different in one more aspect. The *Secret* task was different from the *Service* task, as it required the participant to decode and encode the important configuration values using the *base64*[3] encoding. Similar to the previous difference, the users were provided with a terminal command that would allow them encoding, and decoding any string in the cheat sheet. Therefore, with this difference the *Secret* task should theoretically take longer to solve.

We believe that although these two tasks are not as easily comparable as the two logging tasks, we have done our best to create as comparable tasks as possible. Each

[3]https://en.wikipedia.org/wiki/Base64

of the misconfiguration tasks has a part that requires a bit more effort which hopefully balances out their differences, but that effort is minimized, as we have provided the participants with a helpful hint on how to solve them. Considering both differences however, we expect the *Secret* task to be only slightly more time intensive.

Since both pairs, logging and misconfiguration tasks, of comparable tasks were somewhat similar, we wanted the participants to avoid using any knowledge gained from solving the previous task. We were not able to think of any previous knowledge they could utilize, but we still wanted to remove any unwanted effects that would complicate our comparisons. Thus, we presented the tasks in a zigzag pattern, i.e. the task ordering was:

- Misconfiguration task 1
- Logging task 1
- Misconfiguration task 2
- Logging task 2

Similarly, we have also wanted to show that it will not matter whether the participants will try the web based tools before or after they use the terminal tools. Therefore, we have asked one of the participants to use the terminal tools first, and then the web based tools, while the other participant had the ordering switched. These changes are far from providing major improvements, but they required very little effort from our side, and they were likely to improve our comparisons at least a bit.

The design of questions was rather simple. Our goal was to understand what problems did the participants encounter during the various tasks, and which did they consider very problematic. Additionally, we wanted to see which functionality did they find helpful, and what was their reasoning for each answer. Ideally though, the participants would express their feedback without us asking the questions, and the questions we have written down were supposed to rather stimulate the users to provide us with feedback. After the participants felt that they did not have any further feedback for us, we continued the experiment. You can see examples of the open ended questions below:

- What features did you find useful in the web based tools, and why?
- Did you notice any shortcomings of the web based tools or the terminal ones, could you explain them?
- Were the terminal tools in some ways better than the web based tools, and why?

Plan of the Experiment. At the beginning of the experiment we have provided the participants with three documents. The first was a document explaining the goals of the experiment, our expectations, how the experiment will be carried out, and definitions of the four tasks. We wanted to make sure that the volunteers are able to complete the tasks without any problems, while also making the instructions flexible enough to allow them to try out their own approaches to solving the tasks.

We have also verbally asked the participants whether they would allow us to record their answers to our questions so that we could later transcribe them, and extract information. Usually, we would have put this information into the paper itself, and asked them to sign it, but with the ongoing pandemic, which forced us to conduct the experiment remotely, and considering that the experiment is rather small, we have opted to simplify the situation, and simply have their verbal agreement.

Both participants' instruction documents were similar except for the ordering of the experiments. You can see one of the two documents in the appendix A.1.

The second document that we gave to the participants was a cheat sheet. It contained connection details for the VM where the experiment was carried out, connection details for our Kibana and Dashboard UIs, short explanation of the various Kuberentes resources, and commands that were necessary to complete the provided tasks. The cheat sheet aimed to improve our ability to compare both volunteers. We wanted the task results not to be dependent on their previous experience with Kubernetes, and to decrease the time the participants would need to find required information on the Internet. They would eventually find it on the Internet, but we did not want the ability to find information on the Internet to influence the speed of debugging.

The third document was a simplified diagram that described the connection of the Kubernetes resources that were used in the experiment. It was supposed to help the participants better understand the environment they were moving in. The diagram was identical to the one you can see in Figure 4.7.

After presenting the project to the participants, letting them read the instruction document, and collecting their approval for recording, we asked the participants to use the commands, mentioned in the cheat sheet, themselves so that they are a bit more familiar with them, and they know what is their output. Afterwards, we have asked them to read the first task, and when they feel ready they can start debugging it.

After the participants successfully solved the debugging task, or after they have already spent 10 minutes on solving it, we have asked them for feedback on the tools. If the participants were not sure what to say, we asked them some of the open ended questions that we have mentioned previously. Afterwards, we asked the participants to read the next task, and the process was repeated, until they had successfully solved the last task.

At the end, we thanked them for their participation, and answered any remaining questions they might have had. The last step was to transcribe the feedback they have provided, and delete the recording. After we had the transcript done, we analyzed it, and extracted any interesting observations that were mentioned.

Regarding the time each participant took to solve each task, the results can be seen in Table 4.1. Surprisingly, both misconfiguration tasks, *Secret* and *Service*, were solved in a very similar time of approximately 7 minutes, when using the terminal tools. Both participants were able to solve their misconfiguration tasks using our web tools by approximately 3 and 4 minutes faster than using the terminal tools.

In case of the logging tasks, the first participant was not able to finish the task using the terminal tools because they ran out of time. When using Kibana though, the participant managed to find the required logs under 4 minutes and 30 seconds. The participant ran out of time during the terminal tool test due to various reasons.

Participants	One	Two
Secret - Terminal tools	7min 11sec	
Service - Web tools	3min 4sec	
Service - Terminal tools		6min 46sec
Secret - Web tools		4min 8sec
Logs - Terminal	Did Not Finish	4min 33sec
Logs - Kibana	4min 27sec	7min 58sec

Table 4.1: Time results of the debugging experiment.

Specifically, they first tried to print the logs, and then forward the output to the *grep*[4] command, which they used to filter out logs that were not from the desired time range. They had to repeatedly update the regular expressions used in *grep*, which meant that each adjustment resulted in printing all of the logs from the last 7 days. This took around 20 seconds, so every adjustment slowed them down considerably. After a few tries though, they printed all of the logs into a file, and started using the *grep* on the file. They eventually decreased the number of log lines in that file to around 1000, and they started going through the file manually using the *less*[5] command.

Unfortunately, they were not as experienced with it, which slowed them down. They have also realized that while filtering the logs, they have accidentally filtered out the logs that they were supposed to find. They attempted to recreate the file, but while filtering out the logs again, they ran out of time, and we have continued onto the questions.

The second participant was different. They did indeed manage to find the relevant logs using terminal tools faster than with the web tools. This was completely unexpected, and in a big contrast with the first participant. The major difference was the participants' experience with the VIM[6] terminal tool. They were following a rather identical approach as the first participant did, but there were a few crucial differences that made all the difference.

The second participant immediately printed the logs into a file, and opened it with VIM. They filtered out the logs based on the timestamp using built-in regular expressions, and started removing redundant logs that were repeating. Eventually, they realized they needed to just search for the text that is usually contained in the error logs, and with a few keystrokes they started cycling between the various error logs. In around 30 second, after they realized it, they found the error message that could correspond to the error we were looking for, and we have confirmed that they have found relevant logs.

On the other hand, the second participant had problems with the Kibana tools as it was confusing to them. Specifically, they mentioned that the order in which the logs were shown was inverted, compared to how the logs are usually printed. In other words the Kibana UI sorted the logs from the newest to the oldest, and started with the newest. Conversely, logs in the terminal were also sorted from the newest to the oldest, however

[4]https://linux.die.net/man/1/grep
[5]https://linux.die.net/man/1/less
[6]https://www.vim.org/

the terminal started with the oldest logs first.

Additionally, they selected a wrong time span for the logs in Kibana, and they also accidentally created a filter for error logs that were not relevant for us. They noticed both of these mistakes only after a few minutes, which meant that they made no progress towards finding the target logs. Considering this, in addition to other time consuming things they were experimenting with, such as trying whether regular expressions were allowed in a Kibana search query, their final time for finding the relevant logs was basically doubled compared to the terminal tools. Interestingly enough though, participant number one mentioned no such problems, nor did they have problems with misconfiguring the Kibana search query like the second participant did.

Regarding our observations, and the extracted feedback from the transcribed answers, we noticed a few interesting observations that we would like to mention in no particular order. Starting with general observations.

Although we have tried our best to design simple tasks, the users still needed some time to understand the Kubernetes concepts. For example, the first time one of the participants read the task description, they were confused as to whether they should be editing pods or deployments, and what names they should use. Arguably though, they have read the cheat sheet, and the document also mentioned the name of the deployment. Therefore, it is possible to expect that in the later tasks, the participants had an advantage because at the beginning they were not completely familiar with the Kubernetes concepts, but later on they have started to understand them much better.

Interestingly, both participants did not always provide positive feedback on features they were using rather extensively, and which were present only in the web based tools. In one case they were using the search function of the Dashboard everytime they wanted to display a different service, instead of listing the resources first, and then choosing from the list. However, they never mentioned that they thought the search function was a useful feature, and unfortunately, we have noticed this detail only later on, thus, we could not ask them about it later on.

Our experiment also confirmed that the speed of debugging, using terminal tools was quite heavily dependent on how skilled the participants were. As we have already described, one participant was rather skilled with using terminal tools, especially with VIM, although they both had experience with terminal tools. The experienced participant's knowledge allowed them to use regular expressions inside VIM, and quickly navigate through the logs, while also filtering out the unnecessary lines.

Compared to the other participant who slowly filtered out the lines that were redundant, until there were around 1000 lines of logs left, and then started to manually scroll through the logs. This naturally resulted in the participant running out of time, while the more skilled user was able to finish their task in under 5 minutes.

In contrast, the participant less skilled with VIM was able to complete the logging task with Kibana in half the time of the other participant. They were able to properly select the correct time range, and they created a query which returned just 12 error logs within the first few minutes. Afterwards they simply scrolled through the exceptions, and found the correct one.

In the area of terminal tools, although the participants were familiar with terminal functionality and bash scripting, there were still certain nuances that complicated the debugging when using the terminal tools. One example that clearly impacted the speed of solving the authentication misconfiguration task, was encoding the correct password into *base64* encoding. Specifically, the first participant used the *echo* command to pass a clear text password into the *base64* encoder.

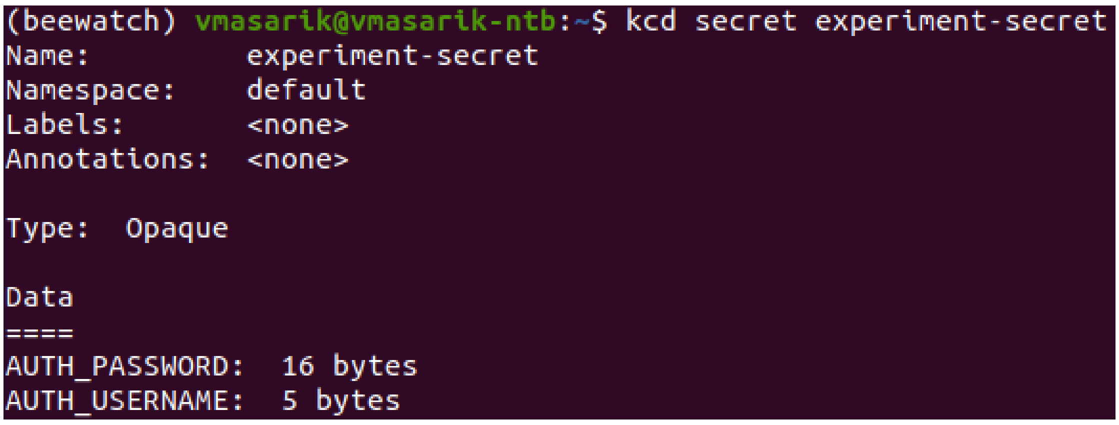
However, the *echo* by default appends a newline character at the end of the text, whenever the text is not enclosed with apostrophes. Therefore, when the participant tried to encode the password, the encoder also included the newline character in the resulting encoded text, which meant that the authentication details were still incorrect. The participant of course did not notice it, and neither did we at the beginning. We have, however, warned them about the possibility of this small detail, and after using apostrophes, they solved the task successfully. Without this knowledge though, the amount of time spent on solving the issue could have easily doubled.

Within the same task, the first participant mentioned that they did not know how to display the configuration details, e.g. the password and username. We expected them to meet with this issue, however we did not expect them to be as confused as they were. They eventually solved it, but they spent a bit of time reading the cheat sheet again, and trying out different commands.

After the experiment, we have tried to find ways of printing the required information that the regular Kubernetes users use. We wanted to find out whether our participant was confused because of incorrect experiment design, or if it was a flaw of the terminal tools. After studying the Kubernetes documentation for a while, we did not find any useful information that would help the participants. The usual ways of printing out details of a Kubernetes resource do not work in this case. As you can see in the Figure 4.8, the default way of describing details using "*kubectl describe*" in the terminal does not allow the user to see the actual data. Even worse this detail is not even mentioned in the documentation. There are two workarounds, however, they are not mentioned in the official documentation either, which makes it hard to learn if the users are not experienced with the Kubernetes[27].

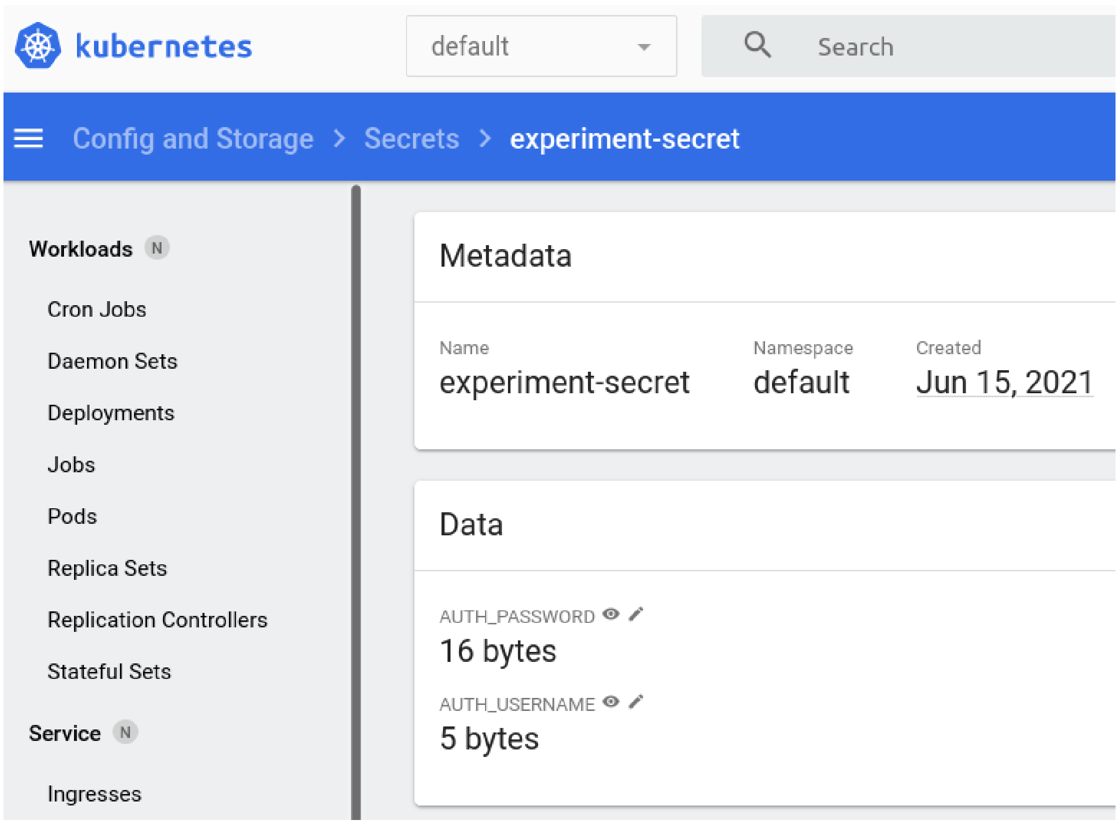
Fortunately, this problem is solved by the Dashboard, as it directly lists the data, and allows you to edit it, as can be seen in Figure 4.9. Moreover, from our observations of the second participant, they had no trouble editing nor listing the details of the configuration.

Lastly, one of the participants was also misled in a surprising way. We have printed an error "*Authentication with the external service failed! Username or Password is incorrect.*" in a pod, to indicate that the application secret details were incorrect. This was part of the *Service* task, and the participant thought it was a response from Kubernetes, which was naturally not true. Unfortunately, the participant did not immediately express their confusion, and we had assumed they understood it is an output of the application. At first they assumed it was part of the experiment, as we did not mention specifically which external service we are connecting to, although that was an unnecessary information. After a minute however, they mentioned that they are not sure about what to do next, and whether it is part of the experiment, to which we have immediately responded, and explained the situation.



```
(beewatch) vmasarik@vmasarik-ntb:~$ kcd secret experiment-secret
Name:         experiment-secret
Namespace:    default
Labels:       <none>
Annotations:  <none>

Type:  Opaque

Data
====
AUTH_PASSWORD:  16 bytes
AUTH_USERNAME:  5 bytes
```

Figure 4.8: Output of the "*kubectl describe*" command used to show details of our authentication configuration.



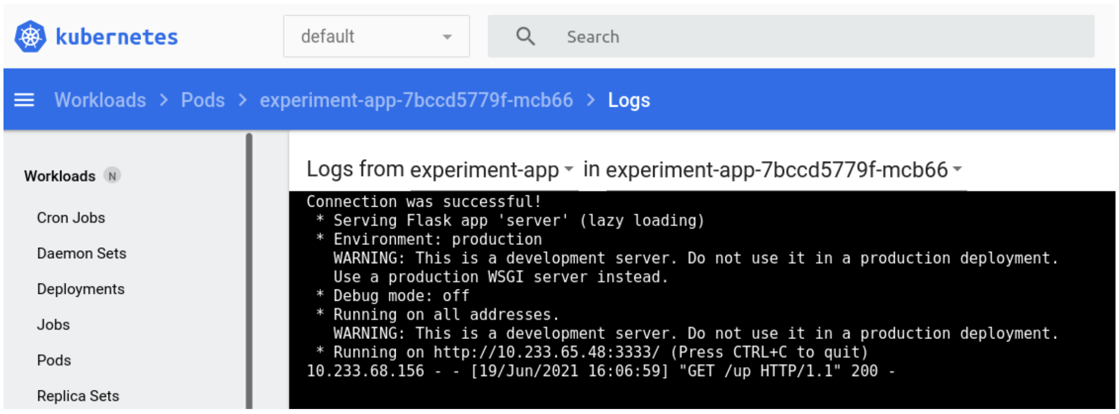
Metadata

Name	Namespace	Created
experiment-secret	default	Jun 15, 2021

Data

AUTH_PASSWORD
16 bytes
AUTH_USERNAME
5 bytes

Figure 4.9: Details of a secret resource listed in the Kubernetes Dashboard UI. The eye symbol next to the variable name allows developers to read its value.


kubernetes
default
Search
Workloads > Pods > experiment-app-7bccd5779f-mcb66 > Logs
Workloads N
Cron Jobs
Daemon Sets
Deployments
Jobs
Pods
Replica Sets
Logs from experiment-app in experiment-app-7bccd5779f-mcb66

```
Connection was successful!
 * Serving Flask app 'server' (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on all addresses.
   WARNING: This is a development server. Do not use it in a production deployment.
 * Running on http://10.233.65.48:3333/ (Press CTRL+C to quit)
10.233.68.156 - - [19/Jun/2021 16:06:59] "GET /up HTTP/1.1" 200 -
```

Figure 4.10: Detail of the pod logs being shown in the Kubernetes Dashboard UI.

In any case, we believe that this was a rather unexpected observation of the experiment. It also shows that even the basics of debugging using terminal tools might be confusing in unexpected ways. We believe this would not have happened in the Dashboard UI, as it clearly shows which text is part of the logs, as can be seen in Figure 4.10.

Conclusion. In summary, we believe that we have shown that our new tools are useful, and that users are able to debug various Kubernetes related issues, especially in terms of misconfiguration. We have encountered many unexpected observations that both confirmed that our web based tools are helpful, and at the same time, we showed that in some cases the web tools might be confusing. Surprisingly, the only negative feedback we have received towards the web tools was with regards to Kibana's UI design, which we cannot adjust. If anything, the experiment showed that terminal tools have multiple negatives, which we have not even considered, and which we are solved by the web tools to some degree. We believe that the goal of the experiment was reached, although in a slightly unexpected manner.

5 Discussion

5.1 Technological Maturity

Kubernetes could be still considered a relatively new technology, as its first commit was in 2014. There are many projects that exist only to support Kubernetes, and have their own maturity[32]. Therefore, they are even less developed, as new software does not include a wide variety of features or a well tested functionality at first. Of course, this should be considered when a new company tries to create a platform that should serve thousands of customers, and they think Kuberentes is the right tool.

During our research, we have realized that our work could be timed very well, as Kubernetes, and projects around it, reached a point where managing applications using Kubernetes might be a viable strategy even for smaller companies. Especially with the development of *managed* Kubernetes products such as *Google Kubernetes Engine*[1] or *Azure Kubernetes Service*[2], where the set up scaling of resources is already automated. It is not a perfect service, and it was not usable in our case, but that situation could be different for many other companies.

Moreover, recently the technologies that simplify management of Kubernetes have become more stable. Helm was updated to a new major version, Nginx Ingress controller[3], Cert-manager[4], and Kubespray documentation is still rather sparse, but there are well known solutions to their current problems, and with a bit more effort, we were able to use them without a problem.

There are still many improvements that can be made, but this thesis proves that it is possible to manage a Kubernetes platform, and reap most of its benefits, by a single developer. This was not always the truth, and as you could see in our section 3.7 on autoscaling, trying to solve not well supported problems ourselves, took a lot of effort. With the technologies constantly improving however, this will not be the case for long, and this thesis could be possibly at the beginning of such a period.

[1]https://cloud.google.com/kubernetes-engine/
[2]https://docs.microsoft.com/en-us/azure/aks/
[3]https://kubernetes.github.io/ingress-nginx/
[4]https://cert-manager.io/

5.2 Limitations

The main limitation that we believe we have encountered was the difficult to compare research on common problems of cloud platforms. The research papers which we have found, all looked into a specific area in which the errors happened, or the source of information was too different to provide a helpful comparison. For example some researchers used publicly available reports[35, 36], some tried data mining public repositories[44], and some monitored the VM themselves[37].

On one hand a research paper[44] tries to classify bugs that occur in cloud systems from multiple perspectives. For us, the important groups seem to be *aspects* and *software*. Although they never explained in detail what these classifications represent, we can assume that the *aspects* group looks at what the bugs have influenced, and *software* means how the bugs are implemented in the code. On the other hand, different research[45] classifies bugs as one of the causes for the cloud incidents, along with misconfiguration or upgrades.

We believe it is not a problem on how the research was conducted, but rather that our situation tries to look at cloud issues in general, rather than in a specific scope. For us, there is no difference whether the network issues are caused on the platform level or whether the application level. If one service cannot reach another, it is a network problem, and we want to know how we can debug such problems in general.

Our goal is to simply find the issues that happen the most in general, so that we can implement tools that can help us in most cases overall, but the research only considers specific contexts. Therefore, we are not able to say which issues happen most often, as we are comparing various contexts.

The result is that we were not able to clearly define categories, nor objectively decide which issues were the most common, nor are we able to say which research should be then considered, and which is irrelevant for our situation. We still believe that our decision to include the most frequently mentioned classifications, and limit the areas to only five, was fairly rational. In particular, we think that our classification includes the most relevant problems, considering that we did not have a unified way of comparing them.

The other limitation that we have met with, was when considering the alternatives in chapter 3. Initially, we planned to consider commercial tools as well. Eventually, we realized that in order to properly assess their benefits, and effort required to implement them, we would have to purchase them, as working with demo versions was not enough. This would of course slow down the research, and we did not want to base our research just on reading the benefits of the tools that the companies listed on their websites.

Moreover, as we were creating a list of commercial tools, and benefits they provide, we have realized that their core functionality does not change. In other words, the core features which we were looking for to solve our issues, were present in all of the commercial tools, and the only difference they had were the various other side features that they offered. Therefore, considering that the core features we were looking for were also included in open source projects, we have decided to limit our search for alternatives to only open source projects.

Lastly, the experiment was not as well designed as we would like to. Naturally, the amount of participants was not ideal, and the knowledge requirements were rather strict as well. We were able to extract very useful information out of our experiment, but having more time, participants, and a wider scope of knowledge and experience for the participants would be very helpful.

Currently, we cannot necessarily say whether our results are objective enough. We have tried our best to avoid any external factors that would influence the results, but with such a short experiment duration, we cannot reliably rule them out. Especially, if we have conducted the experiment for the first time, so we are not sure what could influence the project, and which factors are detrimental to the resulting debugging speed.

5.3 Future Work

During our research, we had to find common issues that affect cloud platforms. We have found quite a few research papers that explored this topic, but comparing many of them was fairly difficult as they explored different areas of bugs, and had different sources of information.

Therefore, although slightly off topic for this thesis, we believe that better understanding of the various contexts in which the bugs happen would be very beneficial for the research community. Especially, it could allow many research papers to be comparable. Currently, the research presents the bugs they have identified, context, and the source of the bug reports. It is clear however that they are very hard to compare, so having research that would create a foundation where we could map these papers, and see the perspectives in which we are able to compare them, would be quite helpful.

We hope this would also help the researchers better map the bugs in different contexts, as currently it seems that they are doing rather based on available data. Thus, it would be useful to know what are the contexts or situations in which bugs can be analyzed. How could the different sources of information influence the classification of bugs, or perhaps, what are the common attributes of bugs or perspectives, based on which we could objectively classify them?

Yet another future topic would be to widen the scope of our questions. Currently, we did research on how small platforms like ours could obtain the most benefits for little effort. Naturally, there are many options we could have explored, and we list a few of them further below, but we did not create a holistic approach that would consider all of them, as it would be incredibly large topic to cover at once. Especially interesting would be to explore concepts which only larger platforms have to consider, such as multi-cluster deployments where the platform consists of multiple Kubernetes clusters. Thus, having an overview of the best practices and tools that improve larger platforms would be quite beneficial.

As mentioned, there are quite a few areas that we wanted to look at in detail, but did not have enough time. Therefore, it could be advantageous to know for example, how creating a database cluster affects the maintainability of a platform. By default, databases are deployed as a single instance, and if a disk, on which the database was saved, failed, we would have lost all the data. Therefore, creating replicas of our database, or perhaps distributing them, would possibly improve the maintainability. There are multiple ways on how to achieve this robustness, but we currently do not know how difficult they are, nor how much maintenance effort they add.

Moreover, as we have discussed how networks can fail, we have identified the tracing of requests through our cluster as a very helpful concept to explore. We even tried implementing it, but in the end it required too much effort for us to implement. Unfortunately, we do not know how helpful it would be in helping us debug problems. Theoretically, knowing the exact path of each request would provide us with incredibly detailed information, but considering that we would be monitoring each request in our network, we expect the tracing to provide us with an even overwhelming amount of data.

Misconfiguration was one of the most common root causes of errors. When researching the topic, we have also discovered the Kustomize[5] project. It should allow us to automate the configuration process even more. Thus, it would further simplify our maintenance efforts, perhaps to such a degree that deploying Kubernetes clusters simply for development purposes would become viable. This would help us in replicating bugs, but also we would be able to test our applications in an environment almost identical to our production environment.

Yet another project which we wanted to implement as part of answering RQ 1.4. was Kubeflow[6]. The project aims to simplify machine learning operations such as training, and use of neural networks. Naturally, for our project this could be very helpful. However, we were not able to find the exact benefits that it would provide us with, and we were also not sure how we would implement such a project into our platform.

[5]https://kustomize.io/
[6]https://www.kubeflow.org/

6 Conclusion

With the modernization of humanity, the biodiversity of ecosystems is at risk. The Beelivingsensor platform utilizes honey bees to act as *living sensors* that are able to estimate biodiversity of the local environments. The platform uses machine learning to extract data from videos, and combine them with other sources to approximate the biodiversity.

The platform has been under heavy development since the middle of 2020. Now, we would like to improve its maintainability and robustness, so that we are able to serve users without major outages. Additionally, as the developers might change over time, we would like them to spend as little effort on keeping the platform operational as possible. Therefore, we will not need to employ a large operations team, and our developers can focus on improving functionality of our applications. Our thesis reaches this goal in multiple aspects, for which we have created explicit research questions.

First, we wanted to see how we can improve the maintenance of our platform. To do so, we have analyzed multiple research papers describing common problems encountered in cloud platforms, and we identified five areas in which we would improve the debugging capabilities of our developers.

Afterwards, we have started looking for tools that would help us with improving the debugging speed. When we have found a few of them, we analyzed them, and decided which we would implement, based on the effort they required to implement, and the benefits they would have brought us.

Second, we wanted our developers to respond faster to any incidents that we might experience. Therefore, we have again searched for multiple tools, analyzed them, and decided which were worth implementing. In this case we have created an alerting system that would immediately notify the developers in case of any unusual patterns, such as an increased number of failing user requests.

Third, we looked at how we could keep our platform operational even when our applications were failing. After implementing our found solutions, we were able to keep multiple clones of our applications running at the same time, and if any one of them would fail, they would be replaced with a new working application. Therefore, we managed to keep our platform accessible to users at all times, even when facing issues.

Lastly, we tried to find any other possible ways of simplifying the work of our developers, which would then allow them to work on more important issues. After our search, we have implemented automated scaling of our hardware resources, therefore freeing developers from having to manually adjust the number of VMs with increasing or decreasing traffic. Moreover, we have also fully automated the deployment of new applications into our user facing environment. This meant that developers no longer had to spend time compiling the applications, and properly deploying them. It also decreased the risk of human error, since the developers now have to only commit new code into our code repository, and decide whether they want the new changes in the production environment.

During the analysis of each solution, we have considered and described multiple alternatives where it was appropriate. Additionally, after each implementation, we have described in detail how exactly the implementation improved the maintainability of our platform. We have also proved the usefulness and functionality of our improvements with two experiments.

One experiment simulated various errors that could happen on our platform, and the result was that all of our users were still able to reach our services. The second experiment, measured how quickly are two developers able to fix misconfiguration issues, and find errors in logs created by our applications. The outcome were a few rather surprising observations, but we believe, we have shown that the newly implemented tools were indeed helpful in solving common problems in our platform.

Overall, we have implemented eight tools that improve the platform maintenance: centralized log collection, external service health monitor, metrics collection, administrative Kubernetes Dashboard, developer notifications, restart of failed applications, automated scaling of resources, and automated deployment of new applications. Naturally, we have made quite a few observations during our work, which we have described in our *Evaluation* and *Discussion* chapters.

In the end, we believe that the main contributions of our thesis are a faster developer response time in case of incidents, simplified debugging of errors, simplified administration of our platform, and improved stability of our operations. Additionally, we have provided a unified overview of common issues with cloud platforms, new suggestions for maintainability improvements, and a helpful guide for smaller teams on which areas of maintainability bring the most value to their platforms.

References

[1] *Common Kubernetes Failures at Scale. https://grapeup.com/blog/common-kubernetes-failures-at-scale/*, Aug. 2020. Accessed: 17.3.2021.

[2] *Summary of the October 22, 2012 AWS Service Event in the US-East Region. https://aws.amazon.com/message/680342/*, Oct. 2012. Accessed: 18.4.2021.

[3] *Network Plugins. https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/network-plugins/*, Mar. 2021. Accessed: 2.3.2021.

[4] *add option to use calico with azure when using calico in vxlan by vanneback |Pull Request #7300 |kubernetes-sigs/kubespray. https://github.com/kubernetes-sigs/kubespray/pull/7300*, May 2021. Accessed: 12.5.2021.

[5] *Settings | Django documentation | Django. https://docs.djangoproject.com/en/3.2/ref/settings/#allowed-hosts*, Mar. 2021. Accessed: 20.4.2021.

[6] *kubernetes-sigs/kubespray. https://github.com/kubernetes-sigs/kubespray*, Apr. 2021. Accessed: 20.4.2021.

[7] *99% of misconfiguration incidents in the cloud go unnoticed. https://www.helpnetsecurity.com/2019/09/25/cloud-misconfiguration-incidents/*, Sept. 2019. Accessed: 20.4.2021.

[8] *Logging Architecture In Kubernetes. https://kubernetes.io/docs/concepts/cluster-administration/logging/*, Apr. 2021. Accessed: 14.4.2021.

[9] *Built-in users | Elasticsearch Guide [7.12] | Elastic. https://www.elastic.co/guide/en/elasticsearch/reference/current/built-in-users.html*, Apr. 2021. Accessed: 14.4.2021.

[10] *kubernetes/kube-state-metrics. https://github.com/kubernetes/kube-state-metrics*, May 2021. Accessed: 10.5.2021.

[11] *Azure status. https://status.azure.com/en-gb/status*, Mar. 2021. Accessed: 20.4.2021.

[12] *Azure status history | Microsoft Azure. https://status.azure.com/en-gb/status/history/*, Mar. 2021. Accessed: 20.4.2021.

[13] *prometheus/node_exporter. https://github.com/prometheus/node_exporter*, May 2021. Accessed: 10.5.2021.

[14] *Open mail relay. https://en.wikipedia.org/w/index.php?title=Open_mail_relay&oldid=996447880*, Dec. 2020. Accessed: 10.5.2021.

[15] *Replication Controller for Kubernetes. https://kubernetes.io/docs/concepts/workloads/controllers/replicationcontroller/*, May 2021. Accessed: 15.5.2021.

[16] *Metrics server issue with hostname resolution of kubelet and apiserver unable to communicate with metric-server clusterIP |Issue #131 |kubernetes-sigs/metrics-server. https://github.com/kubernetes-sigs/metrics-server/issues/131*, May 2021. Accessed: 25.5.2021.

[17] *Docker overview. https://docs.docker.com/get-started/overview/*, June 2021. Accessed: 1.6.2021.

[18] *unable to fetch pod metrics for pod - x509: certificate signed by unknown |Issue #133 |kubernetes-sigs/metrics-server. https://github.com/kubernetes-sigs/metrics-server/issues/133*, May 2021. Accessed: 25.5.2021.

[19] *x509: certificate signed by unknown authority |Issue #146 |kubernetes-sigs/metrics-server. https://github.com/kubernetes-sigs/metrics-server/issues/146*, May 2021. Accessed: 25.5.2021.

[20] *Kubernetes Autoscaler. https://github.com/kubernetes/autoscaler/blob/02985973c6001dbd650b940eef893e4f03db15db/cluster-autoscaler/cloudprovider/azure/README.md*, Apr. 2017. Accessed: 12.5.2021.

[21] *Document that CA is not responsible for registering new nodes by VladMasarik |Pull Request #4092 |kubernetes/autoscaler. https://github.com/kubernetes/autoscaler/pull/4092*, May 2021. Accessed: 30.5.2021.

[22] *Azure Service Health | Microsoft Azure. https://azure.microsoft.com/en-us/features/service-health/*, Mar. 2021. Accessed: 20.4.2021.

[23] *Production-Grade Container Orchestration. https://kubernetes.io/*, June 2021. Accessed: 1.6.2021.

[24] *litmuschaos/litmus. https://github.com/litmuschaos/litmus*, June 2021. Accessed: 10.6.2021.

[25] *Litmus Architecture |Litmus Docs. https://docs.litmuschaos.io/*, June 2021. Accessed: 10.6.2021.

[26] *502 Bad Gateway - HTTP | MDN. https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/502*, June 2021. Accessed: 10.6.2021.

[27] *Secrets. https://kubernetes.io/docs/concepts/configuration/secret/*, June 2021. Accessed: 15.6.2021.

[28] *VM PowerOff Experiment Details |Litmus Docs. https://docs.litmuschaos.io/*, June 2021. Accessed: 10.6.2021.

[29] *Kubernetes |Pod Lifecycle. https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/*, June 2021. Accessed: 10.6.2021.

[30] *Understanding Denial-of-Service Attacks | CISA. https://us-cert.cisa.gov/ncas/tips/ST04-015*, Apr. 2015. Accessed: 24.3.2021.

[31] *Dav4 and Dasv4-series - Azure Virtual Machines. https://docs.microsoft.com/en-us/azure/virtual-machines/dav4-dasv4-series*, Mar. 2021. Accessed: 24.3.2021.

[32] *kubernetes/community. https://github.com/kubernetes/community*, July 2021. Accessed: 5.7.2021.

[33] *Major Microsoft Cloud Outage Blamed on DNS Failure. https://redmondmag.com/blogs/the-schwartz-report/2013/11/outage-blamed-on-dns-failure.aspx*, Nov. 2013. Accessed: 5.5.2021.

[34] *What is Kubernetes? https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/*, June 2021. Accessed: 1.6.2021.

[35] R. Banerjee, A. Razaghpanah, L. Chiang, A. Mishra, V. Sekar, Y. Choi, and P. Gill, *Internet Outages, the Eyewitness Accounts: Analysis of the Outages Mailing List*, in Passive and Active Measurement, J. Mirkovic and Y. Liu, eds., vol. 8995, Springer International Publishing, Cham, 2015, pp. 206–219. Series Title: Lecture Notes in Computer Science.

[36] T. Benson, S. Sahu, A. Akella, and A. Shaikh, *A First Look at Problems in the Cloud*, HotCloud, (2010), p. 7.

[37] R. Birke, I. Giurgiu, L. Y. Chen, D. Wiesmann, and T. Engbersen, *Failure Analysis of Virtual and Physical Machines: Patterns, Causes and Characteristics*, in 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, June 2014, pp. 1–12. ISSN: 2158-3927.

[38] H. Chen, W. Dou, Y. Jiang, and F. Qin, *Understanding Exception-Related Bugs in Large-Scale Cloud Systems*, in 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), Nov. 2019, pp. 339–351. ISSN: 2643-1572.

[39] L. Chen, *Continuous Delivery: Huge Benefits, but Challenges Too*, IEEE Software, 32 (2015), pp. 50–54. Conference Name: IEEE Software.

[40] N. El-Sayed and B. Schroeder, *Reading between the lines of failure logs: Understanding how HPC systems fail*, in 2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), June 2013, pp. 1–12. ISSN: 2158-3927.

[41] K. Fiveash, *AWS outage knocks Amazon, Netflix, Tinder and IMDb in MEGA data collapse. https://www.theregister.com/2015/09/20/aws_database_outage/*, Sept. 2015. Accessed: 18.4.2021.

[42] D. Flanagan, *Javascript: the definitive guide*, 2013.

[43] J. Greig, *Techrepublic Article | DivvyCloud report | Cloud misconfigurations cost companies nearly $5 trillion. https://www.techrepublic.com/article/cloud-misconfigurations-cost-companies-nearly-5-trillion/*, Apr. 2021. Accessed: 20.4.2021.

[44] H. S. Gunawi, T. Do, A. L. A. Sono, M. Hao, T. Leesatapornwongsa, J. F. Lukman, and R. O. Suminto, *What Bugs live in the Cloud? A Study of Issues in Scalable Distributed Systems*, Usenix.org, 40 (2015), p. 7.

[45] H. S. Gunawi, M. Hao, R. O. Suminto, A. Laksono, A. D. Satria, J. Adityatama, and K. J. Eliazar, *Why Does the Cloud Stop Computing?: Lessons from Hundreds of Service Outages*, in Proceedings of the Seventh ACM Symposium on Cloud Computing, Santa Clara CA USA, Oct. 2016, ACM, pp. 1–16.

[46] K. V. Kulkarni, *A brief overview of the Container Network Interface (CNI) in Kubernetes. https://www.redhat.com/sysadmin/cni-kubernetes*, Mar. 2021. Accessed: 3.3.2021.

[47] H. Liu, S. Lu, M. Musuvathi, and S. Nath, *What bugs cause production cloud incidents?*, in Proceedings of the Workshop on Hot Topics in Operating Systems, Bertinoro Italy, May 2019, ACM, pp. 155–162.

[48] D. Love, *The FBI Is Investigating Hacker Group 'Lizard Squad' Over Xbox Live And Playstation Network Attacks. https://www.ibtimes.com/fbi-investigating-hacker-group-lizard-squad-over-xbox-live-playstation-network-1768302*, Dec. 2014. Accessed: 24.3.2021.

[49] M. Shahin, M. Ali Babar, and L. Zhu, *Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices*, IEEE Access, 5 (2017), pp. 3909–3943. Conference Name: IEEE Access.

A

Appendix

A.1 Timed Debugging Experiment

In this section you can find the document used to orient the participants during the Timed Debugging session.

Thank you for participating in our experiment. The core goal is to see how users not experienced with Kubernetes are able to use tools that are available, and whether our newly implemented tools are able to help these users work with orientation and operation. We would also like to collect any feedback from your side that you might have with respect to the newer or older tools. You are not expected to know the Kubernetes concepts, nor tools, and we want exactly the feedback from users that meet for the first time with such tools. So anything you might think, whether you feel that it is biased, we would like to know.

The experiment is divided into 4 stages or tasks. Each stage has a time limit of 10 minutes, so that we keep this experiment short. The stages and their goals are described below. The tasks are not meant to be difficult, but rather see the usability of the tools. We will also provide you with a “cheat sheet” that includes commands that might be helpful for you. You will have some time before the experiment starts to familiarize yourself with the cheat sheet. There is no correct way of solving the tasks. We will start with task one, and continue towards task four. Please let me know when you think you have reached the goal of each task. After each task we would like to ask you a few open ended questions to better learn about your experience.

<say>
In order to accurately capture your feedback, we would like to record your speech when answering questions so that we can later transcribe it, and extract the feedback. The recordings will be anonymized, and as soon as we have transcribed the voice recordings we will delete them. Do you agree with your voice being recorded?

If you are okay with it, could you also share your screen so that I can guide you if you get lost? It is not necessary though.

I have sent you the task document, cheat sheet, and resource diagram, are you able to open all of them?

Please read the cheat sheet, and let me know when you familiarized yourself with it, or whenever you want to continue. Then I will explain the architecture diagram I sent you. Also, feel free to try the commands out so that you know what the outcome of each of them is.

Do you have any questions so far?

Okay, feel free to read the first task, and start whenever you want, or ask questions.
<say>

<Familiarize them with the Dashboard UI>
Task One - Dashboard

- One of your colleagues has created an application called “experiment-app-7bccd5779f-jct5s”.
- This application is trying to authenticate with an external service, but it is failing to do so, you can see its logs.
- Your goal is to make sure that the app can connect to the external app successfully. There should be appropriate logs when that happens.
- The app takes the authentication details out of the secret called “experiment-secret”.

- Username should be “admin”, password “icanholdmybreath”

<Familiarize them with the Kibana UI>
Task Two - Kibana

- One of your colleagues was using an application “web-app-55f5fbdcbb-knc7t”, and they have encountered an error when they tried to add in a new beehive object into the database, which already existed.
- The error occurred a few days ago, and your colleague said it happened on 7th of June at 2:15pm, give or take 30 minutes, they are not sure.
- You don't know what the error was, and you would like to know it, so that you can start debugging your code.
- Your goal is to find the error that might be corresponding to what your colleague described.
- Notes:
    - You know that each exception / error starts with “Traceback (most recent call last):” text.
    - Unfortunately, your application is not well written yet, and there is a “main.models.apiary.Apiary.DoesNotExist: Apiary matching query does not exist.” error periodically reappearing. This is not the error that you are looking for.

Task Three - Terminal

- You are trying to deploy an application “experiment-client-7667f4dcf8-z27l9”, and it needs to communicate with your other application “experiment-app-7bccd5779f-jct5s”.
- To successfully connect you should be able to execute a ping and curl for URL “http://experiment-app:3333/up”.
- This however fails when you log into the pod (kc exec -it <pod name> -- bash) and try running ping or curl to that URL.
- Your goal is to find the error, fix it, and prove that you can curl “http://experiment-app:3333/up”
- Notes:
    - Use “http://experiment-app:3333/up” for curl, but “experiment-app” for ping.

Task Four - Terminal

- Once again, one of your colleagues was using the application “web-app-55f5fbdcbb-knc7t”, and they have encountered an error when they tried to add in a new beehive object into the database, which already existed.
- This error occurred at a different time, and there are other exceptions that you are **not** looking for.
- Your colleague said it happened on 8th of June at 8:30pm, give or take 30 minutes, they are not sure.
- You don't know what the error was, and you would like to know it, so that you can start debugging your code.
- Your goal is to find the error that might be corresponding to what your colleague described.
- Notes:
    - You know that each exception / error starts with “Traceback (most recent call last):” text.

- Unfortunately, your application is not well written yet, and there is a “main.models.apiary.Apiary.DoesNotExist: Apiary matching query does not exist.” error periodically reappearing. This is not the error that you are looking for.

List of Figures

List of Tables