Primož Kočevar

Analysis and implementation of cloud-based architectures enabling software product quality.

Magistrsko delo

Mentor: prof. dr. Andrej Kos
Somentor: prof. dr. Sahra Sedigh Sarvestani

Ljubljana, 2019
Zahvala

Zahvaljujem se mentorju prof. dr. Andreju Kosu za vso pomoč in usmerjanje pri magisterski nalogi in organizaciji obiska v ZDA. Zahvaljujem se tudi moji so-mentorici prof. dr. Sahra Sedigh Sarvestani, ki me je zelo prijazno gostovala v ZDA in pomagala oblikovati magistersko nalogo. Posebna zahvala gre moji družini za neizmerno podporo tekom celotnega študijskega procesa.
Content

1 Introduction
   1.1 Significance ............................................ 5
   1.2 Research objectives ................................... 9

2 Theoretical Foundation .................................. 11
   2.1 DevOps software development practices ................. 11
      2.1.1 Continuous delivery ................................. 14
   2.2 Cloud computing ....................................... 15
      2.2.1 Platform as a service .............................. 20
      2.2.2 Application containers ............................ 22
      2.2.3 Resource Orchestration ............................ 24
   2.3 Distinctions of this research .......................... 26

3 Implementation of the architectures ..................... 29
   3.1 Elastic Beanstalk on AWS .............................. 30
      3.1.1 Elastic Beanstalk ................................. 33
      3.1.2 GitLab ............................................ 37
      3.1.3 Delivery automation .............................. 39
      3.1.4 AWS configuration ................................. 43
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2 Kubernetes on Digital Ocean</td>
<td>51</td>
</tr>
<tr>
<td>3.2.1 Docker</td>
<td>53</td>
</tr>
<tr>
<td>3.2.2 Kubernetes</td>
<td>57</td>
</tr>
<tr>
<td>3.2.3 Delivery automation</td>
<td>61</td>
</tr>
<tr>
<td>3.2.4 Digital Ocean and Kubernetes configuration</td>
<td>71</td>
</tr>
<tr>
<td>4 Results</td>
<td>77</td>
</tr>
<tr>
<td>4.1 Deployment pipeline metrics</td>
<td>77</td>
</tr>
<tr>
<td>4.2 Cloud system snapshots</td>
<td>82</td>
</tr>
<tr>
<td>4.3 Platform neutrality</td>
<td>85</td>
</tr>
<tr>
<td>4.4 Comparison</td>
<td>86</td>
</tr>
<tr>
<td>5 Conclusion</td>
<td>89</td>
</tr>
<tr>
<td>5.1 Limitations</td>
<td>90</td>
</tr>
<tr>
<td>5.2 Further research</td>
<td>90</td>
</tr>
</tbody>
</table>
List of Figures

1.1 A comparison between a single physical server and a cloud-based architecture. .................................................. 7

2.1 NIST conceptual reference model [1]. ........................................... 17

2.2 Cloud Services available to different Cloud Consumers defined by NIST [1]. .................................................. 18

2.3 Cloud provider usage in percentage from all respondents (a single respondent can use multiple different cloud providers) [2]. .......................... 19

2.4 Cloud services in a software stack model. ................................. 20

2.5 Comparison of application virtualization and OS virtualization. . 23

2.6 Resource orchestration operations during the lifecycle of an application [3]. .................................................. 25

3.1 Technological foundations overview. ...................................... 29

3.2 A cloud architecture based on AWS Elastic Beanstalk PaaS and resource orchestration service with a GitLab based deployment pipeline. ........................................... 31

3.3 Elastic Beanstalk architecture of instances [4]. ....................... 34

3.4 Elastic Beanstalk basic workflow[4]. .................................... 35

3.5 Workflow of the deployment pipeline for AWS EB. .................... 40
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.6</td>
<td>Most important configuration options of the EB environment.</td>
<td>46</td>
</tr>
<tr>
<td>3.7</td>
<td>A cloud architecture based on DO managed Kubernetes PaaS and resource orchestration service with a GitLab based deployment pipeline using Docker containers. Static files hosted by the AWS S3 Bucket.</td>
<td>52</td>
</tr>
<tr>
<td>3.8</td>
<td>Docker OS level virtualization [5].</td>
<td>54</td>
</tr>
<tr>
<td>3.9</td>
<td>Docker client-server architecture [6].</td>
<td>55</td>
</tr>
<tr>
<td>3.10</td>
<td>Container architecture compromised of layers [6].</td>
<td>55</td>
</tr>
<tr>
<td>3.11</td>
<td>Kubernetes stack of abstractions for a production environment.</td>
<td>57</td>
</tr>
<tr>
<td>3.12</td>
<td>Kubernetes high level architecture with a master and a worker node.</td>
<td>59</td>
</tr>
<tr>
<td>3.13</td>
<td>Workflow of the first version of the GitLab-Kubernetes deployment pipeline.</td>
<td>62</td>
</tr>
<tr>
<td>3.14</td>
<td>Workflow of the second version of the GitLab-Kubernetes deployment pipeline.</td>
<td>66</td>
</tr>
<tr>
<td>4.1</td>
<td>Success of the pipeline that deploys to AWS during the last year.</td>
<td>78</td>
</tr>
<tr>
<td>4.2</td>
<td>The results of the implemented automated deployment pipeline in GitLab for deploying to AWS.</td>
<td>79</td>
</tr>
<tr>
<td>4.3</td>
<td>The results of the partially automated implemented deployment pipeline in GitLab deploying to Kubernetes.</td>
<td>80</td>
</tr>
<tr>
<td>4.4</td>
<td>The results of the implemented deployment pipeline in GitLab for deploying to Kubernetes.</td>
<td>81</td>
</tr>
<tr>
<td>4.5</td>
<td>CPU utilization in percentage of processing power being used across all cores for a droplet running in a Kubernetes cluster.</td>
<td>82</td>
</tr>
<tr>
<td>4.6</td>
<td>Load average for a droplet running in a Kubernetes cluster. The Droplet uses 1 vCPU.</td>
<td>83</td>
</tr>
</tbody>
</table>
4.7 RAM usage in percentage of all available RAM for a droplet running in a Kubernetes cluster. The measured droplet uses 2GB of RAM. .......................... 84

4.8 Public internet bandwidth over the public internet interface that connects the Kubernetes cluster to the internet. .................... 84

4.9 CPU utilization in the percentage of allocated AWS EC2 compute units that are currently in use on the instance supporting the Elastic Beanstalk application. .......................... 85
## List of Tables

1. List of acronyms ........................................... xiii

2.1 Comparison of performers according to metrics [2] .................. 13

2.2 Comparison of time spent between different performers [2] ........ 14

3.1 Configured DNS records in the AWS web interface using Route 53 (domain.com is used as an example domain). AWS sets TTL values by default. .................................................. 44

3.2 Configured DNS settings in DO web interface using Domains (domain.com is used as an example). DO sets TTL values by default. 72

4.1 AWS deployment pipeline duration comparison ...................... 79

4.2 Kubernetes deployment pipeline duration comparison ............... 81

4.3 Comparison of different architectures for web platform hosting, assessed on the scale from 0-5. ............................................. 87
List of Tables
List of used Acronyms

Table 1: List of acronyms

<table>
<thead>
<tr>
<th>Name</th>
<th>Acronym</th>
</tr>
</thead>
<tbody>
<tr>
<td>International Organization for Standardization</td>
<td>ISO</td>
</tr>
<tr>
<td>Infrastructure as a Service</td>
<td>IaaS</td>
</tr>
<tr>
<td>Platform as a Service</td>
<td>PaaS</td>
</tr>
<tr>
<td>Software as a Service</td>
<td>SaaS</td>
</tr>
<tr>
<td>Relational Database Service</td>
<td>RDS</td>
</tr>
<tr>
<td>Continuous Integration</td>
<td>CI</td>
</tr>
<tr>
<td>Continuous Deployment</td>
<td>CD</td>
</tr>
<tr>
<td>Secure Sockets Layer</td>
<td>SSL</td>
</tr>
<tr>
<td>Time To Live</td>
<td>TTL</td>
</tr>
<tr>
<td>Application Programming Interface</td>
<td>API</td>
</tr>
<tr>
<td>Central Processing Unit</td>
<td>CPU</td>
</tr>
<tr>
<td>Secure Shell</td>
<td>SSH</td>
</tr>
<tr>
<td>Transmission Control Protocol</td>
<td>TCP</td>
</tr>
<tr>
<td>User Datagram Protocol</td>
<td>UDP</td>
</tr>
<tr>
<td>Hyper Text Transfer Protocol</td>
<td>HTTP</td>
</tr>
<tr>
<td>Random Access Memory</td>
<td>RAM</td>
</tr>
<tr>
<td>National Institute of Standards and Technology</td>
<td>NIST</td>
</tr>
<tr>
<td>Amazon Web Services</td>
<td>AWS</td>
</tr>
<tr>
<td>Digital Ocean</td>
<td>DO</td>
</tr>
<tr>
<td>Elastic Beanstalk</td>
<td>EB</td>
</tr>
<tr>
<td>Acronym</td>
<td>Definition</td>
</tr>
<tr>
<td>------------------------------------------------</td>
<td>--------------------</td>
</tr>
<tr>
<td>Content Delivery Network</td>
<td>CDN</td>
</tr>
<tr>
<td>DevOps Research and Assessment</td>
<td>DORA</td>
</tr>
<tr>
<td>Information Technology</td>
<td>IT</td>
</tr>
<tr>
<td>Mean Time To Recover</td>
<td>MTTR</td>
</tr>
<tr>
<td>Operating System</td>
<td>OS</td>
</tr>
<tr>
<td>Quality of Service</td>
<td>QoS</td>
</tr>
<tr>
<td>Elastic Load Balancing</td>
<td>ELB</td>
</tr>
<tr>
<td>Relation Database Service</td>
<td>RDS</td>
</tr>
<tr>
<td>Domain Name System</td>
<td>DNS</td>
</tr>
<tr>
<td>Command Line Interface</td>
<td>CLI</td>
</tr>
<tr>
<td>Web Server Gateway Interface</td>
<td>WSGI</td>
</tr>
<tr>
<td>Canonical Name Record</td>
<td>CNAME</td>
</tr>
</tbody>
</table>
Povzetek

Visok nivo kakovosti programskega produkta pri spletnih rešitvah je ključen pri doseganju visokega zadovoljstva uporabnikov in poslovnega uspeha. Spobnost hitrega odzivanja na zahteve trga in uporabnikov je še posebno pomembna. Za doseganje visoke kakovosti spletnega programskega produkta je zelo pomembna kakovostna programerska implementacija. Vseeno pa je lahko ta neznatna, če je ne podpira primerna programska in strojna infrastruktura. Ta naloga se osredotoča na infrastrukturni del in opisuje dve različni infrastrukturni arhitekturi, ki bazirata na oblačnih storitvah in podpirata implementacijo aplikacije spletnega robota.

Naloga najprej predstavi pretekle raziskave na področju oblačnega računalništva in DevOps principov ter uporabi zaključke teh raziskav kot osnovo za implementacijo dveh dejanskih arhitektur predstavljenih v drugem delu. Na koncu so predstavljeni rezultati dveh implementiranih arhitektur, ki kažejo, da je proces namestitve nove programske kode veliko lažji in hitrejši, če upoštevamo DevOps principe in avtomatiziramo celoten proces. Sposobnost infrastrukture, da se prilagodi na spremembe internetnega prometa je tudi opazno boljša v primerjavi s starejšimi serverskimi infrastrukturami. Interoperabilnost implementiranih arhitektur je opisana na koncu.

Ključne besede: Računalništvo v oblaku, DevOps, orkestracija virov, AWS, Kubernetes, Docker, kontejnerji.
Abstract

Achieving a level of product quality in web software is crucial for satisfying the users and reaching success. Especially an ability to respond to user feedback and market development as fast as possible proved to be very important. To achieve a level of software product quality, the coding implementation is very important, however it can be irrelevant if it is not run on top of the appropriate infrastructure.

This thesis focuses on the second part, describing two different cloud-based infrastructure architectures that support an implementation of a web-scraper application. The thesis first presents related research on using cloud computing technology and DevOps principles and uses the conclusions as a basis for the second part where it describes an actual implementation of the two proposed architectures.

At the end, results of running implemented architectures show the deployment process is considerably easier and shorter after applying cloud and DevOps principles by automating the deployment process. The ability to scale is also considerably better when compared to more classic server-based architectures and the interoperability of the solutions is discussed.

**Key words:** Cloud, DevOps, Continuous Delivery, Resource Orchestration, AWS, Kubernetes, Containerization.
1 Introduction

1.1 Significance

In today’s fast-changing business and technology environment it is very important that a product can quickly adapt to user demands and new technologies. More specifically, a modern web application should be able to quickly release new features, fix bugs, scale and be as independent of the underlying technologies as possible. To be able to quickly adapt to user feedback and integrate new features can be crucial for the usability and growth of a solution that is based on software. Growth of an application is usually already considered a success; however, it brings up a question of scalability. It is crucial to scale the solution appropriately in the case of growth or the usability of a product can be diminished. Additionally, it is important to make the application as independent of the underlying technologies as possible which can bring many benefits when switching providers and hiring new people.

Big scale applications already implement a lot of mentioned principles and reap the benefits of doing so [7] [8] [9] [2]. However, when inspecting the way small to middle-scale solutions implement their application architecture there is a lack of such principles in most cases. That is not very surprising as usually when projects get started it is easier to set up a simple server or one Virtual Machine (VM) and run the code without thinking about scalability, portability or fast release cycles. It depends on the use-case and growth of the application, however, generally, this kind of short-term assembly could bring a lot of problems.
in the long run and even demand a restructuring of the whole architecture as is usually the case when projects grow. Restructuring is usually a costly and time-consuming task and would best be avoided. By using a couple of best practices coupled with the appropriate technologies problems can be partially avoided and save a lot of time and money as well as providing a more feedback-responsive application from the very beginning.

Feedback loops are a crucial component of almost every designed system. In nature, systems or beings that can detect a change and adapt to it the quickest are usually the ones that are most successful and survive. In business and technology, feedback loops are also important as the failure to adapt to market demands can result in failures in business [10]. Feedback loops define how flexible and adaptable a certain system is to external or internal change that usually correlates to better system performance [11]. More specifically in software architectures, fast feedback is arguably the most important asset that can bring usability, resilience, and scalability. An important aspect of software quality is platform neutrality (sometimes referred as interoperability) as it provides independence from platform and infrastructure providers. Interoperability can be reached by using tools that are standardized and open-sourced so that they can be used independently of the company offering them. Additionally, this brings portability, which means the same code can be developed by a wider range of developers that are already used to certain open-source technologies and the same architecture can work with different providers.

A correctly designed architecture can only provide the basis for the goal of achieving a scalable, resilient, usable and interoperable web platform. This thesis will mostly focus on these four attributes of software quality [12] as they seem the most improved by implementing the right architecture. It must be noted that it is not enough to use the right architecture, but the implementation on top of that architecture is also crucial. For example, if a very resilient architecture is combined with a very unreliable implementation on top of it, the goal of resilience will not be met. Only the correct combination of the right architecture and
implementation for a distinct use-case brings the attributes of software quality [9]. It is also very important that each architecture is adapted to the business use-case as a very successful solution can completely fail when used for a different use-case.

Figure 1.1: A comparison between a single physical server and a cloud-based architecture.

Observing the attributes of software quality in real-world platforms, usually, a compromise between them must be met. For example, a very small and non-scalable platform can easily implement resilience as it can enable a lot of backups and redundancy. However, when the same application grows and requires scalability, resilience is more difficult to achieve. Similarly, a smaller platform is much more developer-friendly and simple than a larger scalable one. Thus, it can be observed that usually a compromise between them should be reached, however,
this again depends on the use-case.

In Figure 1.1 comparison between the single server architectures and cloud-based architectures can be observed. A cloud-based architecture implements multiple complex parts of the system and integrates them in a single platform. That brings simplicity and manageability to the whole architecture as the whole system can be easily controlled using one platform. Most crucial for the significance of this thesis are the feedback loops presented and the difference between the speed of them.

The loop going back to code presents an ability of the software to quickly adapt to changes in the interaction of users with the platform which can result in new features and new software bugs. The same is usually possible when dealing with the single server architecture, however, the difference is the speed with which the cloud-based architecture can deploy the changes to production and thus make the feedback loop much faster which is indicated by a thicker line in Figure 1.1.

The loop going back to the architecture is even more important as it presents the ability of the architecture to respond to the events that happen because of the interaction between the users and the platform. These events can be unexpected behavior of internet traffic (e.g. a change of weather produces growth if the web platform is a weather channel), new software bugs and external unexpected events that affect the implemented software (e.g. a used application programming interface (API) behaving unexpectedly or giving false data). In the case of the cloud-based proposed architecture, it can respond to such events automatically, if configured correctly. Architecture implemented without such mechanisms such as the one on the left side in Figure 1.1 is usually much slower in responding as the feedback loop depends on the engineer (usually system administrator) that has to change the attributes of the architecture manually, which can take from hours to weeks. Quality and speed of the response an architecture has on such events defines how scalable, resilient and efficient it is.
1.2 Research objectives

The original research contribution of this master thesis is the implementation analysis of novel concepts to small scale cloud architectures enabling a compromise between different attributes of software product quality. This thesis uses a definition of software product quality by ISO 25010 standard [12] that defines eight quality characteristics. More specifically it focuses on scalability, resilience, portability and usability provided by the infrastructure architectures that support web platforms. Crucially, this thesis does not present a general solution to achieving a high level of software product quality, as the methods of reaching this goal vary widely depending on the use-case and business needs of the implemented platform [9]. Furthermore, an architecture implementation that works for one company or team of developers, will prove unusable for another, as reaching mentioned goals depends heavily on the way a platform is implemented on top of the architecture. Therefore, this thesis provides the tools for setting up the fundamental architecture of tools that enable a compromise between scalability, resilience, usability, and portability, however, an exact implementation on top of that architecture that suits the use-case finally determines if these positive attributes can be reached. Thus only results of the designed architecture are presented and not the whole web platform on top of it, as this would be out of scope for this thesis.

This thesis compares the classic single physical server architecture to novel cloud-based architectures, presented in Figure 1.1. Most importantly, it compares the speed and agility of feedback loops the architectures provide and compares how that affects the chosen attributes of portability, usability, resilience, and scalability, which, among others, define software quality [12].

Work for this thesis is mostly distributed in two parts, that present most of the work done outside of writing. The first part is the analysis of related research in the field of software quality, DevOps and cloud computing in Chapter 2. Different concepts in these fields are presented from a theoretical perspective as well as the
benefits and disadvantages they bring. The findings of this research analysis give a good basis for the implementation presented in the second part.

The second part of the work is presented in Chapter 3 and describes an actual implementation of two different cloud-based systems for running an architecture of services supporting a web platform, both in providing resources and building a deployment pipeline for the code. Technologies used to implement these architectures are introduced and the implementation of these technologies is described in detail. One of the main purposes of Chapter 3 is to provide documentation of the work done, such that a reader could reproduce the implementation and achieve similar results, as a need for such an architecture is often recognized.

The results of the two implementations are compared with the goals set by the research and compared with a more traditional single-server architecture in Chapter 4. In the end, according to the results, appropriate conclusions are made in Chapter 5.
2 Theoretical Foundation

In this chapter, a theoretical foundation behind the later presented implementation is described. An overview of related research in topics important for this thesis is presented and gained knowledge from research is used as a basis for the implementation in Chapter 3. The theoretical foundation is presented in two parts and extracts research insight into the benefits and risks of different approaches. Firstly, an overview of novel concepts relating to better software product quality and organizational performance using DevOps principles is presented in Section 2.1. As a part of DevOps development practices, an insight into continuous deployment is provided in Chapter 2.1.1.

The second part in Section 2.2 presents a theoretical analysis of technologies used to achieve DevOps principles while using the Cloud Computing model. A definition of the Platform as a Service (PaaS) is given in Section 2.2.1 and its usage is considered. Orchestration and containerization are presented as two crucial concepts used in scalable, resilient and portable cloud architectures. At the end, in Section 2.3, distinctions of this research are presented as a way of building upon gathered knowledge from studied research.

2.1 DevOps software development practices

DevOps is an understood set of practices and cultural values that have been proven to help organizations of all sizes improve their software release cycles, software quality, security, and ability to get rapid feedback on product development [2].
In this Chapter, a short description of DevOps principles is provided. More importantly, a set of findings, how DevOps affects organizations, is presented. For this purpose, research done by Dr. Nicole Forsgren and DevOps Research and Assessment (DORA) group is considered [2]. For deeper analysis and further reading, a book from both Nicole Forsgren and Jez Humble should be considered as it provides the latest research in this field and provides a lot of additional information on how the research was conducted [13].

Discussing the metrics and groups used to compare performance in software delivery in the Accelerate: State of DevOps report 2018 [2]. Actors delivering software are classified into low, medium and high performers. Hierarchical cluster analysis is used to classify actors into groups, based on a level of software delivery performance, measured in throughput and stability. Thus, actors in one group are statistically similar to each other in terms of stability and throughput, and different compared to other groups. Ward’s method is used for clustering based on the change in fusion coefficients, several individuals in each cluster (solutions including clusters with few individuals were excluded), and univariate F-statistics. There is no differentiation between industries, level of regulation or size when classifying. This means that high level and low level of performance can be achieved across all industries. It also shows that usually similar practices lead to good results independent of the type of industry or level of regulation, which can be against intuition.

Two basic attributes of throughput and stability are discussed. Throughput is measured through deployment frequency and change lead time. Deployment frequency is measured by number of deploys to production per year. Change lead time is measured in minutes of time that takes the developers to put new code to production. Stability is measured through Mean Time To Recover (MTTR) in hours and Change failure rate in percentage of changes that produce failure. Results between different performers can be seen in Table 2.1.

As can be observed in Table 2.1, 2018 report shows a new group forming
as a subset of high performers, indicated as elite performers. Elite performers
group is small but represents an improvement over the growing number of high
performers group. This is good, as it means that there are more organizations
performing similarly to high performers and even a subset that is performing
better than observed before. However, it is concerning that the gap between
software delivery performance of low performers and high performers is growing,
which means low performers are lagging.

Table 2.1: Comparison of performers according to metrics [2]

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Elite</th>
<th>High</th>
<th>Medium</th>
<th>Low</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deployment frequency</td>
<td>on-demand</td>
<td>hourly-daily</td>
<td>weekly-monthly</td>
<td>weekly-monthly</td>
</tr>
<tr>
<td>Lead time for changes</td>
<td>&lt;1 hour</td>
<td>day-week</td>
<td>week-month</td>
<td>month-6 months</td>
</tr>
<tr>
<td>MTTR</td>
<td>&lt;1 hour</td>
<td>&lt;1 day</td>
<td>&lt;1 day</td>
<td>week-month</td>
</tr>
<tr>
<td>Change failure rate</td>
<td>0-15%</td>
<td>0-15%</td>
<td>0-15%</td>
<td>46-60%</td>
</tr>
</tbody>
</table>

As can be expected, high performers achieve much lower change failure rate
and their time to restore service is considerably less compared to low performers.
However, compared to medium performers such big difference is not observed.
The most unexpected result is that elite and high performers achieve much better
throughput as well, compared to low and medium performers. This result is
quite counterintuitive as throughput and stability were once considered a trade-
off, as it was supposed that to reach stability, big releases had to be thoroughly
tested in a lengthy process before they were deployed to production. Research
results presented in the DevOps report 2018 [2] prove that for the big majority of
applications higher throughput and stability go hand in hand to provide better
software delivery performance. This discovery acts as a basis for implementing
CD and implementing a lot of concepts described in this thesis as an alternative
to older architectures that were once considered to be effective.
2.1.1 Continuous delivery

Continuous delivery (CD) is a software engineering approach in which teams keep producing valuable software in short cycles and ensure that the software can be reliably released at any time [14].

CD describes implementation details of how to achieve better stability and throughput as described in the previous Section 2.1. This approach was proven to deliver lower risk releases, as releases are smaller and thus present lower risk. CD also provides an opportunity to speed up time in which software is delivered to market, meaning the latest features and solved bugs can be released faster. An important aspect of implementing CD is an improvement of team satisfaction as it makes releasing software easier and thus reduces team burnout. Frequent releases also prove more engaging for developers as the feedback from users is faster and they feel more connected to their work [8] [15].

Table 2.2: Comparison of time spent between different performers [2]

<table>
<thead>
<tr>
<th>Time spent</th>
<th>Elite</th>
<th>High</th>
<th>Medium</th>
<th>Low</th>
</tr>
</thead>
<tbody>
<tr>
<td>Development</td>
<td>50%</td>
<td>50%</td>
<td>40%</td>
<td>30%</td>
</tr>
<tr>
<td>Configuration Management</td>
<td>5%</td>
<td>10%</td>
<td>30%</td>
<td>30%</td>
</tr>
<tr>
<td>Testing</td>
<td>10%</td>
<td>20%</td>
<td>50%</td>
<td>30%</td>
</tr>
<tr>
<td>Deployments</td>
<td>5%</td>
<td>10%</td>
<td>30%</td>
<td>30%</td>
</tr>
<tr>
<td>Change Approvals</td>
<td>10%</td>
<td>30%</td>
<td>75%</td>
<td>40%</td>
</tr>
</tbody>
</table>

Implementing an architecture that deploys at the push of a button has additional benefits when looking at the amount of time the team can now spend on more important things compared to dealing with the deployment architecture. The improvements can be best seen in Table 2.2 where time spent for each group of performers can be observed. It is obvious that elite and high performers have more time for development which means they have more time for innovation, implementing new features and improving the software generally. That additi-
onal time for development can lead to considerable business and technological advantages compared to competitors.

Good results of Continuous Delivery (CD) combined with Continuous Integration (CI) can be seen in the case of HP LaserJet Firmware team [16]. This is not an apparent case for the implementation of CD practices as they were building firmware, which is usually much harder to deploy than web software and thus makes the case for implementing these practices even stronger. Their change lead time changed from one week to 3 hours and the number of commits changed from 1 commit/day to 100 commits/day.

Implementing DevOps principles, with a lot of automation in their pipeline they achieved an improvement of 700% in time spent on developing new features and considerably lowered the time spent on integration and maintenance. This kind of improvement can fuel innovation and growth.

For more information on CD, a book that invented the term should be considered [17].

2.2 Cloud computing

Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable communication and computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction[18].

There are many definitions of cloud computing, however, the definition provided by National Institute of Standards and Technology (NIST) seems most in line with the work done in this thesis thus the model provided is used to classify different technologies that are used. There are a few essential characteristics that define if a used model is indeed using cloud computing principles:
• **On-demand self-service** - computer resources can be automatically provisioned.

• **Broad network access** - functions can be accessed through mobile phones, computers and tablets.

• **Resource pooling** - provider resources are pooled dynamically from a shared pool of resources, of which customer has no control.

• **Rapid elasticity** - quantity of capabilities are elastically provisioned up and down depending on the need of the consumer.

• **Measured service** - resources are continuously monitored and transparently reported.

These are also the principles the described architecture is based on and tries to accomplish.

In Figure 2.1 a conceptual reference model is provided by NIST and presents an overview of all the different entities that are involved in providing a service to a cloud consumer. In this thesis, a role of a cloud consumer will be represented as the two implemented architectures that use two different PaaS providers. It can be observed that there are a lot of different actors such as Cloud Auditors that take care of security, privacy and performance audits that are crucial. Cloud Provider also usually takes care of business support, provisioning, portability, service orchestration and lower-level layers dealing with hardware. All these services would be difficult to replace when using a single server architecture where a customer (an implementer of the architecture) would have to take care of this completely separately. For a big majority of use-cases, clear benefits of the cloud computing model can be noticed. It must be observed that a disadvantage of using cloud computing services is a possible vendor lock-in where a customer would depend on the provided services so much that it would be impossible to change the vendor. However, using the right architecture, this can be mitigated and is discussed later in this thesis.
Figure 2.1: NIST conceptual reference model [1].

Cloud computing model is composed of 3 different service models that crucially differentiate the way cloud services are used by cloud consumers. The differences can be best observed in Figure 2.2 as three different cloud consumers are presented according to the services they use. This thesis focuses on a PaaS consumer perspective as it is arguably the best option when trying to implement a web platform on top of service while trying to achieve all mentioned attributes. As an example of a PaaS service, Amazon Web Services (AWS) Elastic Beanstalk (EB) and Kubernetes on Digital Ocean (DO) are used in this thesis. They both enable easy application deployment, integration and database support.

In comparison, a Software as a service (SaaS) consumer would have higher-level services available like document management, content management, and others. Implementation of these kinds of functionalities by the consumer would not be necessary, however, it also brings a compromise considering the freedom of
development and customization is limited. An example of these kinds of services are Google Apps, Dropbox or Salesforce.

In contrast to SaaS, there is also an option of Infrastructure as a service (IaaS), which provides the most customizable service as it is the closest to hardware and has the least amount of abstractions. It is almost like renting physical servers and hardware that is scalable and provides rapid elasticity with some added features like backups, service management, content delivery network (CDN)… However, a lot of features like integration and application deployment, which are already included in a PaaS, need to be implemented additionally. This additional step demands new skills and time, which sometimes makes especially smaller teams (<10 engineers) skip it and lose a lot of important functionalities in their architecture. However, for consumers with very specific and unique needs, IaaS
should be considered. Widely used IaaS examples are Amazon EC2, Digital Ocean Droplets, and Google Compute Engine.

A choice of cloud computing and PaaS is also suggested by the latest research done by DORA [2]. They report that customers that meet all five essential characteristics of cloud computing are 23 times more likely to be elite performers. Research also shows that consumers that use platform as a service to host their applications are 1.5 times more likely to be in a group of elite performers.

In research results observed in Figure 2.3, it is clear that cloud computing is used by a big majority of respondents. Specifically, 67% of respondents said that their primary application is hosted on cloud. Additionally, there are predictions that cloud usage will grow to 83% by 2020 [2]. It is worth noticing that all the results together, bring more than 100%, which is at first counterintuitive. However, this is a consequence of the same respondents using multiple different
cloud providers and thus classifying in multiple groups at once. Usage of multiple different providers is good for interoperability, portability and availability of the application.

At the end of this chapter, it should be mentioned, those good properties can be achieved also without using cloud services. Good results were observed when DevOps practices and cloud concepts were correctly applied to mainframe architectures with physical machines. However, this usually demands much more effort, time and knowledge.

2.2.1 Platform as a service

Platform as a service is a category of cloud computing services that provide application developers with an ability to deploy, manage and develop without the knowledge of infrastructure management, provisioning and other complexities usually associated with running an infrastructure architecture for a production-ready application.

Figure 2.4: Cloud services in a software stack model.
Looking at the classic software stack notation in Figure 2.4, PaaS is placed in the middleware layer, as opposed to the Operating System (OS) layer and Application layer. Middleware layer provides software building blocks that are used by PaaS consumers (e.g. software libraries, databases, virtual machines...), however, they can also usually be configured for a specific need. Middleware layer usually must be installed by IaaS costumers that operate on the OS layer and have a choice of more operating systems and libraries to install.

It can be much easier for an organization to achieve a production ready architecture for running and managing the application. Usually, PaaS services still allow for a lot of customization and configuration on an infrastructure level. For example, in AWS EB, a consumer can still configure and manage EC2 instances that are running beneath EB and thus has almost all of the control an IaaS consumer would have. However, there are some cases where a lower level IaaS architecture would have its benefits, as PaaS is usually developed for a preferred number of technologies and programming languages. Thus, in some cases, if a consumer wants a very customizable architecture, that can run a wider number of technologies IaaS is preferred. It must be mentioned, that tools such as Kubernetes, that are also offered as PaaS (Google Cloud, Digital Ocean...), are being developed with containerization in mind and can run almost anything that can be containerized. This will be explained more in detail in sections that follow.

Interoperability of services is the capability to communicate, execute programs, or transfer data among various cloud services under specified conditions. Portability can be considered in two different aspects as data and system portability. Data portability is the ability to transfer data from one system to another without being required to recreate or re-enter data descriptions or to modify significantly the application being transported. System portability is the ability of a service or an architecture to run on different types and sizes of cloud providers [1].

As a PaaS customer, it is important to think of interoperability and portability
properties of the chosen architecture and provider. A portable and interoperable architecture at the beginning of a project, when this does not seem so important, can prove very useful in the long run. Consider a case where a web platform is completely dependent on one PaaS provider and the technologies of that cloud provider do not allow for any interoperability or portability. This means a platform is not flexible to use any other technologies and providers that may be better at doing certain specific things, as the architecture is not interoperable with other providers. Additionally, if a used provider abuses the lack of portability and a good position in the market with charging higher prices, it can prove fatal for the business model of the imagined web platform as the lack of portability forces them to stay with the provider and pay the price.

2.2.2 Application containers

Application containers such as Docker are based on application virtualization technology. Virtualization, in general, is defined as the emulation of the software and/or hardware upon which other software runs [19]. Application virtualization technologies are named thus because they usually contain only one application or even a very specifically defined function of an application, which provides the basis for the implementation of the microservices architecture. Using the microservices architecture is one of the ways to achieve attributes mentioned in this thesis, but this thesis only provides the tools and basic concepts to implement such an architecture but does not focus on implementation, as that would be out of scope. Application virtualization technologies mainly focus on providing an automated, portable and reusable way of deploying applications [20].

The implementations of application virtualization in this thesis are referred to as application containers or containers, as this is the name that is accepted in the industry and academia. The implementations of OS-level virtualization are thus referred to as Virtual Machines (VMs). A decision to use containers over VMs can be best understood by comparing the two.
In Figure 2.5, the biggest difference between the two can be observed, considering a VM runs lower in the software stack compared to a container and uses a Virtual Machine Manager (Hypervisor) to run on top of hardware, that is virtualized from the VMs point of view. Each VM needs an installation of the complete OS that runs on top of the Virtual Machine Manager with drivers and libraries that are needed to use the virtualized hardware.

Alternatively, a container acts as a lightweight OS on top of the host OS (e.g. Windows or Unix) using the container runtime to communicate with the host OS which is shared between all the running containers. Consequently, containers can use shared resources (storage, memory, processor...) and therefore be much more resource-efficient compared to the VMs, which results in a much faster boot-up time (few minutes compared to few seconds) [21]. Additionally, containers usually already include the code for running the app and all the needed dependencies defined in the config file, as shown in Figure 2.5.
Operationally, containers are very different from VMs. Traditional VMs are usually booted up, configured, maintained and upgraded through the lifetime of the application. That means they have a state that changes with time. Containers take a completely different approach and try to achieve statelessness while implementing the concept of configuration as code. A container is usually defined in a configuration file that determines what will happen when the container is booted up and defines an immutable state of the container. When a change is needed, practice is for an old container to be destroyed and a new container set up with the updated configuration file. This presents a very declarative and immutable way to run and maintain applications.

The use of application containers provides a basis for implementing more advanced architectures using orchestration services, which sometimes use containers as basic building blocks of their system (e.g. Docker and Kubernetes) and combined provide an efficient way to run multiple services. Boot up speed, immutability and the declarative nature of containers provide portability and consistency. The same container can be deployed either on a developer’s laptop or a production server and moving the development environment to a different machine can be much easier.

### 2.2.3 Resource Orchestration

When talking about the usage of cloud computing, classical system administration work of configuring and connecting servers to achieve some level of quality of service (QoS), is now replaced with the operation of resource orchestration. Resource orchestration is defined as a set of operations that cloud providers (e.g. AWS, DO) and application developers (this is the presented role in this thesis) undertake (automatically or manually) for selecting, deploying, monitoring and dynamically controlling the configuration of hardware and software resources. The configuration defines a system of components (e.g. databases, servers, load balancers...) that provide the required level of QoS and deliver the application
2.2 Cloud computing

Figure 2.6: Resource orchestration operations during the lifecycle of an application [3].

Resource orchestration supports the application implementation so that it can achieve high QoS if implemented correctly. Figure 2.6 shows resource orchestration is done in multiple steps during the lifecycle of the supported application to provide the best QoS.

Firstly, resources are selected based on the predicted usage type and amount of load (e.g. a scraper application implemented in Django, with low expected traffic in the beginning and expected slow growth). Next, resources are connected, configured and deployed, running the code of the application (e.g. Django python code). When the application is running in production, monitoring of the resources is crucial and thus presents the next step in resource orchestration (measurements of hardware can predict and detect a failure and different anomalies). According to the results of the monitoring, resources need to be controlled and configured...
(e.g. in the case of growth and higher load, more computing power is needed). If resources control cannot provide enough support, new resources need to be selected and therefore the cycle starts again.

Importance of the speed of feedback loops was already discussed in this thesis and it is no different in this case. If the architecture can move through the described cycles as fast as possible or preferably run them concurrently, it signifies good resource orchestration and high QoS.

2.3 Distinctions of this research

This thesis implements the concepts presented by related research to provide a unique perspective on implementing big-scale modern paradigms to small scale web platform architectures. It builds upon the findings and good practices presented in this chapter by implementing them in two architectures of services supporting web software and deployment pipelines. The goal of doing this is to provide infrastructure support for a higher quality software product when combined with the correct software implementation. The exact software implementation on top of the architecture is also mentioned, as it is also very important to achieve high quality, however, this is out of the scope of this thesis and is not discussed in detail.

More specifically, an implementation of two different PaaS based architectures, supporting a web scraper platform, are described in detail in Chapter 3. The cloud architecture that is based on AWS technologies and services is described in Section 3.1. The second alternative architecture that is using Kubernetes as an orchestration engine and Digital Ocean as a provider is described in Section 3.2. Cloud technologies and DevOps principles are used as a foundation when implementing described architectures. Implementation is presented more from an infrastructural standpoint; therefore, the software programming part of the solution is omitted. Finally, the results of the implemented architectures are
2.3 Distinctions of this research

presented in Chapter 4 and split in two parts. Firstly, the results of the implemented deployment processes (pipelines) are aggregated and compared across different implementations in Section 4.1. Second part in Section 4.2 describes the results of the infrastructure that is running the code and supporting the software platform with storage, networking, computing and other resources. In the end, appropriate conclusions and comparisons regarding the results of the implementations are made in Chapter 5.
3 Implementation of the architectures

This chapter describes the implementation of two different cloud-based PaaS architectures. Both are implemented for a similar purpose of supporting web applications. An implementation of each architecture is presented in two parts. Firstly, an overview of most important technologies and tools used in the actual implementation is presented. This is important for a general understanding and
motivation behind the usage of these tools.

Secondly, an implementation of the designed architecture is described and the actual code for the configuration is provided and interpreted. Problems that were encountered while implementing are described and solutions to most of the problems are described. Therefore, the implementation part can present a useful read for anyone dealing with such tools and architectures.

The final implementations of the configurations might look quite simple, however multiple iterations of the configurations were tested and implemented, which were usually more complex, but did not work as well. This was observed everywhere, in Dockerfile configuration, Kubernetes configuration and GitLab jobs configuration. Usually the simplest solutions were discovered at the end and worked best, however all the prior attempts are necessary for the discovery of the final solution. Not all the attempts and implementations were presented in this thesis, as it acts as a type of documentation of the final architecture.

3.1 Elastic Beanstalk on AWS

Architecture described in this section is composed of two bigger blocks as seen on Figure 3.2. First block presents a deployment pipeline that enables continuous deployment and automation combined with version control. Automated tasks are usually triggered by the arrival of new code to the selected branch. Here, Git as a tool for version control is used, as it is considered as the leading tool in this field. This pipeline enables a lot of customization and configuration; therefore any kind of automation can be implemented, from integration to deployment. The implementation of a scraper application described in this thesis does not demand the implementation of automated testing and thus this part was omitted. A different and more resilience focused application (e.g. an application for a bank), could demand a different approach and automated testing would be a necessary part of the pipeline.
Figure 3.2: A cloud architecture based on AWS Elastic Beanstalk PaaS and resource orchestration service with a GitLab based deployment pipeline.
Second block contains the whole AWS infrastructure of services provided as a PaaS, named Elastic Beanstalk. Elastic Beanstalk also acts as a resource orchestration tool that aggregates all the necessary components provided by AWS on the infrastructure level (e.g., EC2, Relation Database Service (RDS) database, Elastic Load Balancing (ELB)...). Out of the box, it provides logging, monitoring and alert services already integrated in the platform. In this architecture of services, EC2 instances act as computing resources that run the code for the application. They are replicated and destroyed according to the needs of the users so that the best efficiency and consistency are ensured. EC2 instances can also be in different availability areas (Europe, Asia, US ...) so that availability and latency is improved. ELB acts as balancer that distributes the load of internet traffic coming from a single domain, to several different EC2 instances that handle the requests and act as servers. To deploy new code to the EC2 instances, a single deploy command is used. The setup and authorization needed for running this command is described in Section 3.1.3 where the whole deployment pipeline is described.

The AWS RDS provides a database service that uses PostgreSQL database and connects to the EC2 instances that communicate with it securely according to the security rules defined in the AWS console. Route 53 acts as a Domain Name System (DNS) web service that routes the requests that come to a specified domain according to the rules set. Different subdomains can be routed to different EB instances or even to servers outside of the AWS.

The information for this implementation detail is mostly supported by official documentations from AWS and GitLab [22] [4]. A lot of other implementation details are gathered from other internet resources like Stackoverflow and GitLab forum.
3.1 Elastic Beanstalk on AWS

3.1.1 Elastic Beanstalk

Elastic Beanstalk is an essential tool for the architecture presented in this chapter. It determines the way how code is deployed to the AWS services and infrastructure, therefore determines the functionality in the second block on figure 3.2. However, the usage of Elastic Beanstalk also determines how code is deployed and managed, therefore also defining the design of the deployment pipeline in the first block.

Elastic Beanstalk is not very easily defined, as there are many different definitions and usage cases for it. Therefore, multiple definitions and perspectives are presented.

It can be considered an orchestration tool for the services and infrastructure provided by AWS. Therefore, in some cases, it can be compared to Kubernetes and other orchestration tools such as Docker Swarm. However, it functions quite differently than those tools, as it does not necessarily use container technology and mainly works with AWS services and can not be run independently on top of on-premise architectures. This makes it maybe a bit simpler to use out of the box, as it already integrates a lot of AWS services like capacity provisioning, load balancing, scaling and application health monitoring.

The other side of Elastic Beanstalk is that it also acts as a sort of a PaaS running on top of AWS infrastructure. Elastic Beanstalk basically provides and abstraction layer above the AWS IaaS components to make them easier to use correctly. There are people that argue Elastic Beanstalk is only an IaaS management tool. However, the line between the two is very thin and for the purpose of this thesis, Elastic Beanstalk is considered a PaaS. That makes it a competitor to services such as Heroku, OpenShift and Engine Yard. Currently AWS supports those services on top of their infrastructure, which has business implications, as the growth of Elastic Beanstalk threatens those services, which are also clients of AWS. Business implications, even if they seem unimportant at first, can be very important for the technology, as they can determine which tool will prevail in the
long run and have the most support.

For this architecture a Heroku as a PaaS was actually considered, and a testing implementation was examined. Heroku provides useful features that are easily implemented, maybe even simpler than Elastic Beanstalk. However, as AWS promotes Elastic Beanstalk and has a big influence in the cloud computing market, it is an obvious choice for a long-run strategy. It is also developed constantly and upgraded so that more and more features are available in the easiest way. Its integration with the AWS infrastructure also usually provides more control of the underlying architecture compared to Heroku.

Figure 3.3: Elastic Beanstalk architecture of instances [4].

Figure 3.3 provides an overview of different building blocks of a general Elastic
Beanstalk architecture. The instances are available through the load balancer in the case of AWS named Elastic Load Balancer that takes care of appropriately balancing the load between different availability zones and EC2 instances. The computing and serving building blocks of the Elastic Beanstalk system are EC2 instances that can be in different auto scaling groups and different availability zones. Auto scaling groups are groups of EC2 instances that must maintain a level of capacity in the group. Several instances in an auto scaling group can be set manually or automatically according to metrics using scaling policies. Usually each availability zone has its own auto scaling group, so that a consistent capacity is ensured in each zone. Availability zones are locations where AWS locates its servers and data centers (e.g. EU - Frankfurt, U.S. East - N.Virginia, U.S. West - Oregon...). They are connected by low-latency connections, but still as independent as possible, so that if one zone goes down (e.g. loss of electricity), the other one is on a different grid and stays up.

RDS instances are the main database building blocks of the Elastic Beanstalk system. They are also distributed through different availability zones for fault tolerance purposes. If one zone goes down, data can still be served by the database in the other one. S3 Buckets, which are AWS main storage service, provide backups of RDS databases and configurations of the whole Elastic Beanstalk environments.

![Figure 3.4: Elastic Beanstalk basic workflow[4].](image)

The workflow of Elastic Beanstalk is presented on Figure 3.4. The actions in the workflow such as creating applications and deploying new versions can be
done either by using an awscli tool or through the AWS web console. Firstly, an application is created usually using the web interface. In this step a name for the application is chosen, a description and different keys specific to this application. Next, an environment is created which defines the type of technologies (Python, PHP, Ruby...) that run on the infrastructure and for what purpose the environment is used (a general server environment or a worker environment). At the end of this procedure, code that will run is selected or a sample app is deployed if code is still developed. New versions of the code are deployed using the awscli console or by uploading the zip file containing the code. This cycle repeats every time new changed to the production code are accepted, while monitoring and managing tools are used on the deployed environment.

However, there are multiple ways an environment can be updated with new code. Considering that the environment may be composed of multiple EC2 instances in different availability zones, the complexity of this operation should be considered. There are four different instance update policies available, each useful for a group of use-cases:

- **All at once** - deployment is done simultaneously for all instances, which produces a short time of unavailability during the deployment. This is the default option in the beginning if you only have one instance available and is also the cheapest on the resources as it does not demand any new instances spinning up.

- **Rolling** - a chosen number of instances is updated at a time so that during deployment capacity of the environment is lower as some of the instances are unavailable. However, the rest of the instances are still available and thus unavailability is probably avoided.

- **Rolling with an additional batch** - very similar to rolling updates, however at the beginning a full batch of instances is replicated to ensure full capacity during deployment.
• **Immutable** - a new batch of instances is started while old batch is still running and the old batch is replaced only if the new update is successfully deployed to new instances. If the update is not successful, no rollback is needed as the old instances are still running.

### 3.1.2 GitLab

The main purpose of GitLab is version control using Git as a version control tool, which is most widely used, open-source version control tool. However, GitLab also provides a lot of lifecycle and deployment tools that can help to enable DevOps concepts in a desired system without the need for other CI/CD tools. GitLab allows the integration of monitoring and continuous deployment by using runners, that act as processes that are triggered at a certain point (usually when something is committed to a specific branch). Some of the most important services provided by GitLab listed and described:

- **Git version control** - Git is used for version control, with added features as issues, monitoring and analysis of the activity of the repository, access control for teams, merge handling... In this sense it acts similarly to the widely used version control alternative service GitHub.

- **CI/CD pipeline** - An implementation of deployment and integration pipelines by using Runners as a computing entity. Any kind of automation and scheduling that can be run on Runners can be implemented. Monitoring of the whole pipeline is enabled.

- **Operations integration** - environments that are deployed can be connected, measured and tracked for errors. There are also options to connect with Kubernetes clusters using Prometheus to monitor and even change the configuration of the clusters using GitLab. Using functions as a service in a Serverless setup is also possible.
• **Container registry** - for storing container images in a place that is not public, but can only be accessed with a key. This is quite useful when using Kubernetes, that needs accessible container images and is also implemented in this thesis in a later chapter when dealing with Kubernetes.

This thesis mostly uses GitLab for version control and CD pipeline integration. GitLab offers the tools to run version control on computer or server for free and open source. GitLab CI/CD runners can also be run on a local machine or servers so that a project can be completely independent of the payed services GitLab provides and some insight on that is provided. GitLab as a company only charges for usage of services like runners and private repositories on their servers.

For this thesis, GitLab is the main tool for running the deployment pipeline and version control demonstrated in the first block in Figure 3.2. Git version control hosting is used as a service provided by GitLab, however self-hosted architectures were implemented for testing as well. Currently, hosting code on GitLab is free of charge for what is needed, however a self-hosted setup can be quickly considered and implemented if prices change. GitLab runners, responsible for running the deployment and testing pipeline, were also tested in a self-hosted environment. However, the final implementation is using a Digital Ocean Droplet to host the runners (it is like an EC2 instance in AWS and acts as a server hosted by Digital Ocean). The implementation of that is almost identical to the one on a self-hosted server. This kind of pipeline is quite flexible and independent of the providers as they can be quickly replaced because of the open-source nature of the tools. There is also a simpler option to use paid runners provided by Digital Ocean, however control over machines running runners is sacrificed if that option is used.

A file that defines the events in the implemented GitLab pipeline is `.gitlab-ci.yml` that is located in the root of the project. It defines jobs and stages as this are the building blocks of the GitLab pipeline. Jobs are the smallest entities in the GitLab pipeline as they define the exact processes that
get executed in GitLab runners. Stages are a higher abstraction (e.g. build, deployment, test) and can contain multiple jobs that can run in parallel. Stages run successively, thus if the build stage fails the test or deployment stage after that are skipped as they usually require the build stage to be successful.

There are also different attributes available in the `.gitlab-ci.yml` file that determines what happens in the pipeline. A `tags` attribute of a specific job defines the type of runner this job can run on, as it has to match the tag of the installed runner. For example, if a tag Docker is used, than a job should run on a runner that has the tag docker which usually means it is installed. If the same job is run on top of the runner that does not have Docker installed it will probably fail.

An `only` attribute in the `.gitlab-ci.yml` defines what version control branches trigger a specific job. A deployment job can be triggered only when code is pushed to the production branch and a build job can be triggered every time new code is pushed to the master. The events can be customized for a specific use-case and automation needs.

### 3.1.3 Delivery automation

The deployment pipeline implemented for deploying code to AWS EB is simple compared to the one for a Kubernetes cluster described later. This section provides an insight into developing such a pipeline with GitLab and how to deal with various challenges that can occur. The pipeline on GitLab is defined with one job in a `.gitlab-ci.yml` file, located in the root of the project. The pipeline described in this section is presented as the first GitLab block in Figure 3.2.

Runners are implemented in a docker container running on a Digital Ocean droplet. In this place, AWS EC2 instance could be easily used, as it acts similarly regarding to the installation of runners. However, for the simplicity of the implementation, both proposed architectures share the droplet with installed runners on Digital Ocean. Digital Ocean droplet is chosen because of the clearer pricing
model, compared to AWS EC2 instances. Additionally, one runner can easily be used for multiple projects, as it is unlikely that two runners will be needed at the same moment. Even if that occurs, a job will wait for a free runner and execute when possible, which should not produce problems.

A used droplet is the smallest instance of the available droplets, using 1GB of RAM memory and 25GB of storage space with 1 virtual CPU core available, which is enough for the current use-case but can be upgraded very simply if needed. The instance is provisioned using the Digital Ocean web interface, the credentials for the Secure Shell (SSH) are provided and used for the connection with the droplet. When connected through SSH, the GitLab runners are installed according to the official documentation [22]. At the last step of the installation when registering the runner with the repository so that it can use it, a Docker executor should be chosen. If the Docker executor is chosen, the choice of an appropriate image is very important, as it represents what kind of dependencies are available to the runner. According to the chosen image, it is useful to tag the runner to know what is running inside (e.g. in the runner that docker:stable image is used, tag is docker and in the other where python alpine:latest image is used, a tag is python). There are a few problems that can be encountered if the runner
3.1 Elastic Beanstalk on AWS

is installed on a local computer but that is analyzed later in this paper, where a fully local setup of GitLab is considered.

A GitLab runner with python installed is used for this pipeline as an execution tool for defined jobs. There is no need for running Docker containers as dependencies are few and can be installed directly using pip commands defined in the `.gitlab-ci.yml`.

Figure 3.5 presents a workflow of events that have to happen for a Django application to get deployed to Elastic Beanstalk successfully. First three blocks of the workflow are defined in the `.gitlab-ci.yml` configuration file and presented in this section. Last two blocks are triggered when code is deployed to Elastic Beanstalk, and are defined in the configuration folder `.ebextensions` and presented in the next section, as they are a part of the Elastic Beanstalk architecture.

The Elastic Beanstalk deploy job uses a production only tag, which defines that only changes pushed to the production branch trigger the deployment pipeline. The deployment job first installs the `awscli` and `awsebcli` tools defined in the `cd/deployReq.txt` file. These are tools to manage AWS resources from the CLI. Any additional dependencies or requirements in the future can be simply added to the `cd/deployReq.txt` file without the need of changing the code of the job.
Implementation of the architectures

Listing 3.1: An Elastic Beanstalk deploy job in the .gitlab-ci.yml

```
1 eb deploy job:
2 stage: deploy
3 tags:
4   - python
5 script:
6   - mkdir ~/.aws/
7   - touch ~/.aws/credentials
8   - pip install -r cd/deployReq.txt
9   - printf "[eb-cli]\naws_access_key_id=\%s\naws_secret_access_key=\%s\n" "$AWS_ACCESS_KEY_ID" "$AWS_SECRET_ACCESS_KEY" >> ~/.aws/credentials
10  - touch ~/.aws/config
11  - printf "[profile_eb-cli]\nregion=ap-southeast-1\noutput=json" >> ~/.aws/config
12  - export PATH=~/local/bin:$PATH
13  - eb deploy
14  - python cd/startWorker.py
15 only:
16   - production
```

Next, AWS credentials in a form of AWS_ACCESS_KEY_ID and AWS_SECRET_ACCESS_KEY are provided from the environment variables of the runner and copied to the hidden file ~/.aws/credentials. These environment variables are set in the web interface of the GitLab repository by the maintainer of the repository only, as they present crucial information that should be kept secret. The location configuration for the AWS tools is copied to the ~/.aws/config hidden file and the PATH variable is updated for the installed tools to be accessible from bash.

The command `eb deploy` uploads the project to the AWS cloud and triggers events that are defined in the .ebextensions folder if the upload is successful. The upload is managed by the installed AWS CLI tools that should have the right credentials for this to work. If there is a problem with the deploy command
at this stage, the whole deployment process is reverted, and the last working deployment is used. A history of deployments is stored in the AWS S3 and they can be restored at any time from the AWS EB web console. When an application is uploaded to EB, configuration defined in the .ebextensions folder is triggered. A very important fact about the EB deployment is, that with every deploy, the instances get build and configured from the start and all previous changes made outside of the configuration are reverted. Thus, the proper configuration should always be done using configuration files and not by configuring manually using SSH. Challenges and solutions regarding are described in the next section.

3.1.4 AWS configuration

The configuration of AWS services and infrastructure is mostly done through the AWS web console or the awscli CLI. Both are used in this architecture; therefore a more general description of the configuration is given, without the specific instructions on which button to press. The part described in this section is presented as the second AWS block in Figure 3.2.

Managing resources using the AWS web console is usually quite easy compared to the CLI set up. The web console is also more powerful and gives more functionalities, compared to the awscli. AWS CloudFormation templates are the way big projects define their AWS infrastructure and is probably the most standardized and compact way. However, no CloudFormation code is discussed in this thesis as it is a bit too complex to grasp for a small project.

It is important to set up an IAM user with the proper credentials when starting to use AWS, so that there is no root account (or the one root account is not used). IAM user credentials that manage the AWS infrastructure should have very specific permissions. For example, if one IAM user credentials are used to push static files to the S3 bucket, they should be different than the IAM user credentials deploying to Elastic Beanstalk. The users and their permissions should be as specific as possible, so that in the case of a breach, it can be quickly
traced back to a specific user, which helps with solving the problem quickly and finding the reason for the breach.

Elastic Beanstalk is used as a main driver of the presented architecture; however some additional AWS services are also used. Route 53 is used as a DNS web service, RDS service is used as a database provider, S3 is used for backups, certificate manager is used to manage Secure Sockets Layer (SSL) certificates of different domains and other services that are indirectly used by Elastic Beanstalk.

A certificate manager takes care of SSL certificates for different domains that are used in the Route 53. Here certificates can be imported or requested. The presented architecture uses Amazon issued certificates that are the simplest to use inside of AWS and free, compared to other offerings on the market. Amazon also automatically renews them when necessary.

Table 3.1: Configured DNS records in the AWS web interface using Route 53 (domain.com is used as an example domain). AWS sets TTL values by default.

<table>
<thead>
<tr>
<th>Type</th>
<th>Hostname</th>
<th>Value</th>
<th>TTL (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MX</td>
<td>domain.com</td>
<td>10 mail.domain.com</td>
<td>300</td>
</tr>
<tr>
<td>A</td>
<td>mail.domain.com</td>
<td>195.206.228.92</td>
<td>300</td>
</tr>
<tr>
<td>A</td>
<td>domain.com</td>
<td>ALIAS domain.eu-central-1.elasticbeanstalk.com.</td>
<td>-</td>
</tr>
<tr>
<td>A</td>
<td>platform.domain.com</td>
<td>ALIAS domain.eu-central-1.elasticbeanstalk.com.</td>
<td>-</td>
</tr>
<tr>
<td>CNAME</td>
<td><a href="http://www.domain.com">www.domain.com</a></td>
<td>domain.eu-central-1.elasticbeanstalk.com.</td>
<td>300</td>
</tr>
<tr>
<td>SOA</td>
<td>domain.com</td>
<td>&lt;zone transfer details&gt;</td>
<td>900</td>
</tr>
<tr>
<td>NS</td>
<td>domain.com</td>
<td>ns-2009.awsdns-59.co.uk.</td>
<td>172800</td>
</tr>
<tr>
<td>NS</td>
<td>domain.com</td>
<td>ns-453.awsdns-56.com.</td>
<td>172800</td>
</tr>
<tr>
<td>NS</td>
<td>domain.com</td>
<td>ns-1140.awsdns-14.org.</td>
<td>172800</td>
</tr>
<tr>
<td>NS</td>
<td>domain.com</td>
<td>ns-969.awsdns-57.net.</td>
<td>172800</td>
</tr>
</tbody>
</table>

Route 53 is a DNS service used for managing owned domains and their redirects. If a domain is owned, the control of the redirects to the domain and the subdomains is controlled from the Route 53 dashboard. For example, if a domain is used for serving a web application, then the base domain is redirected to the Canonical Name (CNAME) record of the Elastic Beanstalk. However, the same
domain can be used for the mail server. In that case the new MX record is needed in Route 53 that will forward all mail server request to the mail subdomain (mail.domain.com). Also, the mail.domain.com subdomain should redirect requests to the mail server (usually an A record with the value of IPv4 of the mail server). This is implemented also in the case of the presented architecture. If there are other platforms that exist on subdomains separately, then a record for a subdomain can be created that redirects the requests to that resource (usually a CNAME or an A record).

Table 3.1 describes a valid DNS record configuration in AWS Route 53 currently running in production. State of Authority records are omitted from the table as they provide information for the transfer of the domain and a lot of details about the zone transfers like server names, transfer times and serial numbers that are required for a domain transfer. Two subdomains used (www. and platform.) use ALIAS records to redirect requests to the base domain. For all records, Time To Live (TTL) setting is left at the default value set by AWS. TTL dictates the amount of time a DNS record is valid. When a resolver gets a new DNS record, it saves its value in cache for the amount of time specified in TTL. After that time has passed, it requests an update and checks if the value has changed. Therefore, regular A and CNAME records use shorter TTL, compared to NS records. That means a change of A or CNAME record will propagate faster through the network, compared to the NS record, as it is kept in cache for a longer time. However, if needed, TTL values can be changed and adapted to the specific use-case.

When first setting up the EB web server environment, the CNAME is set and a preconfigured python platform is chosen. Most of the widely used platforms/languages are available, however if there is a need for something specific, there is an option for a custom platform. A custom platform functionality is not tested in this thesis and thus can not be discussed, however if there is a need for a custom platform, maybe an infrastructure level (CloudFormation of EC2 instances) solution should be considered as it can provide more customizability. When a platform/language is chosen for an EB environment, it is difficult to change
later and that should be considered. At this point the initial code that is first
ly uploaded to the EB environment is chosen but can be overwritten later with
the deployment of new code. From the point of creating the EB environment,
properties of the deployed environment are set in the configuration overview.

The configuration overview is the most important part of the EB environment
for a system administrator or a DevOps engineer, in the web console it can be
found as a configuration tab. It configures all the resources associated with the
Elastic Beanstalk environment and connects them. Configuration overview, pre-
sented in Figure 3.6, defines the configuration of most tools and the environment
as a whole. Configuration of the EB environment can also be saved to S3, if the
configuration needs to be replicated in the future. However, if the environment
depends on external resources, perfect replication is difficult to achieve as those
resources would also need to be replicated.

![Figure 3.6: Most important configuration options of the EB environment.](image)

Database support is provided using RDS on top of the PostgreSQL engine.
Recovery and backup capabilities are already integrated in this database. To connect the database to the Elastic Beanstalk instance, it must be configured from the configuration tab in the EB environment. If the database is created from there, all other connections to the database endpoint are refused for security purposes. If external access to the database is needed, new security rules should be configured on the selected database instance by configuring a new security group and attaching it to the RDS instance. This can be useful when doing external backups or moving data, but also presents a possible security issue in the future.

The size of EC2 instances running beneath EB is set in number of processes and threads in the configuration. The capacity is set to 4 maximal instances and at least one instance to run the application. Different metrics such as network traffic in/out, CPU utilization, latency, different disk metrics and health of instances can determine when instances are scaled.

Configuration also connects an EB environment load balancer to the SSL certificate in the certificate manager. Load balancers can be chosen between application, network and classic load balancers according to the use-case of the server. For example, an application load balancer can redirect mobile or API URLs to a different instance in the environment. In this architecture, classic load balancer is used as it is most simple and suits the needs of the application.

To start deploying a folder with source code to the Elastic Beanstalk application, `eb init` command is run inside the root folder of the project. For this command to be successful, awscli and awsebcli packages have to be installed and proper credentials provided for deploying and initializing an application like discussed in the previous section in the deployment pipeline. The initialization requires a default region to be defined, the name of the application that will be used (or a name for a new application), a platform version used (e.g. Python 3.5, PHP 5.5) and an SSH keypair that is used for accessing the instances. After initialization, a folder containing a file with all this information is created in the
root of the project (.elasticbeanstalk/config.yml).

Listing 3.2: An EB application information file in .elasticbeanstalk/config.yml used in this architecture.

```yaml
branch-defaults:
default:
environment: <applicationName>
group_suffix: null
production:
environment: <applicationName>
global:
application_name: <applicationName>
branch: null
default_ec2_keyname: <keyname>
default_platform: Python 3.6
default_region: eu-central-1
include_git_submodules: true
instance_profile: null
platform_name: null
platform_version: null
profile: eb-cli
repository: null
sc: null
workspace_type: Application
```

When git is used as a version control tool, there are a few more options available. An important difference, compared to using EB without git, is that the code in the latest commit to a specified git branch is deployed and not what is currently in the folder. The result is that every deployment is connected to the specific commit, which can prove useful when looking back to see exactly what was deployed. This can be confusing at first, if switching from a different version control tool like Subversion, which was also experienced during the implementation of this architecture. The `branch-defaults:` key specifies from which branch a deployment is allowed and is currently set to production as discussed.
To configure the EB environment from the code repository configuration files in the .ebextensions folder are used. They should have a .config extension and can be formatted in JSON or YAML. For this configuration YAML is used. Advanced configuration is provided, as under the key option_settings properties of AWS resources can be changed. The configuration options mentioned under the configuration overview in Figure 3.6 can all be set using the option_settings key. Another section of a .config file that is important and used in this architecture is container_commands. Here are specified all the commands that are run at the beginning of deployment when underlying EC2 instances are being set up. The configuration section container_commands should include all commands that are usually applied to a server using SSH manually, as those are disregarded when a new deployment is applied. Nginx is a proxy server used by EB and can be configured using container commands in the beginning of deployment if that is needed. Container commands should be applied and defined sequentially by tagging them with numbers in the sequence. This way of configuration presents a more immutable infrastructure that is set at the beginning and not changed much later during the lifetime of the deployment. Using SSH to login to an instance should only be used for debugging purposes or one-time operations and not for installing dependencies. There are a lot of other sections available (packages, sources, files, users, groups, commands, services) and can be useful for OS level configuration of underlying EC2 instances.

The last two blocks of the deployment workflow presented in Figure 3.6 are implemented using configuration in the .ebextensions folder.
Implementation of the architectures

Listing 3.3: Specifying container commands in .ebextensions/db-migrate.config file

```python
container_commands:
  01_migrate:
    command: "django-admin migrate"
    leader_only: true
option_settings:
  aws:elasticbeanstalk:application:environment:
    DJANGO_SETTINGS_MODULE: project.productionSettings
```

The migration part of the deployment pipeline is defined in the .ebextensions/db-migrate.config file and is executed after the code is successfully uploaded to the EB, but before the Nginx server is started. Leader only tag means that the migration is only applied once in the case of replicated nodes (servers). In this file, additional container commands can be used for setting up the database or storage before the server is running (e.g. creating administrator users, collecting static files...). The last line in this file is very important, as it specifies which settings are used for running the container commands. During the development of this architecture, problems were encountered, because wrong settings were used by running the `python manage.py migrate` command instead of the `django-admin migrate`. The `django-admin` command requires settings to be specified and can avoid a lot of issues with using the wrong settings.

Listing 3.4: Specifying a wsgi script for running a server in .ebextensions/django.config

```python
option_settings:
  aws:elasticbeanstalk:container:python:
    WSGIPath: project/wsgi.py
```

The last file .ebextensions/django.config is crucial for running the application, as it specifies the path to the wsgi.py file in an environment variable. A wsgi.py script defines the Web Server Gateway Interface (WSGI) for the
Nginx server running the application. Django provides a simple implementation of WSGI, that defines the type of settings used in the `wsgi.py` file. This is very important, as settings used in `wsgi.py` are usually production ones, compared to the local development settings used in the `manage.py` file. This way of using settings can be useful, as there is no need to change between settings when running in development, where `manage.py` is run and therefore development settings are automatically used. These details, however, are already very specific to Django and are not discussed further.

### 3.2 Kubernetes on Digital Ocean

Second architecture, described in Figure 3.7, is again constructed from two blocks. However, now these blocks are more complicated and contain more different entities compared to the AWS architecture. That is because this architecture does not depend on a single provider and uses more interoperable tools such as Kubernetes and Docker. It also includes more compromises, AWS S3 was used for serving static files instead of DO Spaces as the AWS alternative proved faster and more reliable. The kind of benefits and disadvantages this approach brings is discussed in the next chapter where results are presented.

First block again presents a deployment pipeline that enables automation and version control. The deployment is again triggered by a push to a git branch (usually production or master). However, as the architecture is based on Kubernetes, the usage of containers is necessary, as they are the basic building blocks of Kubernetes clusters. Docker is chosen for the task of containerization as a most well established and open-source tool for that purpose. Therefore, the list of needed dependencies and procedures are defined and together with the source code, a Docker image is built. For this image to be used, it must be pushed to a registry that is accessible by the Kubernetes master. Uploading the source code to the second block is done in two steps as static files are hosted separately by the AWS S3 Bucket. Static files are uploaded by simply using an AWS sync
Figure 3.7: A cloud architecture based on DO managed Kubernetes PaaS and resource orchestration service with a GitLab based deployment pipeline using Docker containers. Static files hosted by the AWS S3 Bucket.
command.

The second part is uploading a Docker image to the Kubernetes cluster on DO. This is done by applying the configuration that is defined in the Kubernetes configuration file and includes the hash of the docker image that includes the source code. After the configuration is applied, Kubernetes master tries to pull the defined image from the docker image registry. For this, it must have access to this repository. According to the configuration of the cluster, the source code is replicated across multiple nodes or only to one node. Kubernetes master takes care of the replication and manages the different nodes that act like servers with Ubuntu 18.04 installed. These nodes can be in different availability zones and their replication and scalability can be configured.

Digital Ocean also provides monitoring, alerting and logging in a very user friendly and easy to set up manner. Load balancer is again provided and acts a bit differently as it forwards all the traffic to the Kubernetes cluster, where a Kubernetes master distributes the load. However, in this case, load balancer also works as a DNS web service and takes care of forwarding request from multiple subdomains. A DO managed database is used instead of AWS RDS. It is again a PostgreSQL database that has backups, monitoring and security already implemented, only left to be configured.

The information for this implementation detail is mostly supported by official documentations from Digital Ocean, GitLab, Kubernetes and Docker [22] [6]. Other implementation details are gathered from different internet resources like Stackoverflow, Docker forum and GitLab forum, where some discovered solutions are also posted.

3.2.1 Docker

Docker as a concept can have different meanings. It can present Docker Inc. a company that is providing Docker as an enterprise solution and a platform. However, in this thesis Docker is used as a software tool for providing operating-
system level of virtualization that can be called containerization. In this thesis the latter definition of Docker as a containerization tool will be used every time Docker is mentioned.

Figure 3.8 describes a simple architecture that is running Docker as a virtualization tool on top of the host OS. This presents benefits compared to the older way of virtualization using VMs.

![Docker OS level virtualization](image)

Figure 3.8: Docker OS level virtualization [5].

The basic building blocks of a Docker container are layers and images. How they are connected and imported is defined in a Dockerfile file that includes the exact sequence of layers in the container. The difference between images and containers is that images are read-only parts of a container that are only used by a container and not edited. Container usually adds a specific top writable layer that uses resources defined and provided by the underlying image. However, a build container can be used locally as an image from another container and additional layers can be added on top. If this built container is pushed to a public registry,
it can be used by others outside of a local environment as well. The architecture of a container based on a public Ubuntu image is presented in figure 3.10. Using official images can remove the long and dangerous act of installing an OS and all needed dependencies manually which can create unpredictable results when done by different entities.

Layered implementation brings big advantages when it comes to efficiency of storage and communication. If a container is pushed to a registry that already
has the needed images and layers downloaded, the push will only sync the needed
layers which can be very small (less than 1MB). This small layers are usually the
ones that are changed during development and configuration and are presented
as read/write (R/W) container layers on figure 3.10. Thus, a new container that
presents a newer version and is in full size quite big (0.5 - 1GB) only requires
small amount of data transfer (20MB) to the server as all the previously used
layers already exist.

However, basing containers on external images provided by a third-party so-

curcise can present a security risk, as the images can contain malicious acting parts.
There is a known case when a third party docker image was used for mining
crypto-currencies for the attackers [23]. Thus, care should be taken when using
external images and, in most cases, official images are preferred as they are less
likely to contain any malicious code.

A storage driver is responsible for the communication between these layers.
There are also a wide variety of storage drivers available and each presents a
certain trade-off. However, any more detail on this topic would be out of scope
of this thesis and more details can be found here [24].

In a way, a Docker container is a bit like a virtual machine. But unlike a virtual
machine, rather than creating a whole virtual OS, Docker allows applications to
use the same Linux kernel as the system that they’re running on and only requires
applications be shipped with things not already running on the host computer.
This gives a significant performance boost and reduces the size of the application
[25].

When developing Docker images and containers, a structure of a Dockerfile is
crucial and thus understanding the basic commands available is very important
to successfully use Docker.

- **FROM** - Specifies the image on top of which the container is based. It is
  usually used in the beginning of the Dockerfile and the image used should
  be accessible to the machine that builds the container from the Dockerfile.
3.2 Kubernetes on Digital Ocean

- **RUN** - executes commands specified in a separate layer and builds a new image. It is often used to install dependencies and software packages supporting the command specified in **CMD** or **ENTRYPOINT**.

- **CMD** - sets a default command when a container runs. Parameters of the command can also be provided but both can be overwritten at the time when the container is run.

- **ENTRYPOINT** - provides a point from which a container will run when it is started. For example if a container should be used to run a specific bash command, that is specified in the **ENTRYPOINT** at the end of the Dockerfile and is used as an executable.

### 3.2.2 Kubernetes

This section describes Kubernetes as an orchestration tool for managing containers and presents concepts that are used for the implementation of Kubernetes. Implementation in this thesis uses a managed Kubernetes service provided by Digital Ocean. Similar Kubernetes platforms are provided also by other cloud providers like Google Cloud, AWS and Azure.

<table>
<thead>
<tr>
<th>Application</th>
<th>• Different containerized applications.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data plane</td>
<td>• Capacity for running and connecting containers (CPU, memory, network, storage)</td>
</tr>
<tr>
<td>Control plane</td>
<td>• Orchestration layer exposing API to manage containers</td>
</tr>
<tr>
<td>Cluster infrastructure</td>
<td>• Provides and maintains VMs, networking, security groups …</td>
</tr>
<tr>
<td>Cluster operations</td>
<td>• Managing the cluster, implementing security, storage, networking, logging, monitoring…</td>
</tr>
</tbody>
</table>

Figure 3.11: Kubernetes stack of abstractions for a production environment.
It is crucial to understand, that Kubernetes by itself is not a PaaS offering, but rather an open source orchestration tool, which makes it a bit different compared to Elastic Beanstalk. However, as mentioned, some providers do offer PaaS services using Kubernetes. This section mostly omits that part and focuses on the Kubernetes architecture and concepts.

Figure 3.11 presents a stack of abstractions in a Kubernetes production system and groups functionalities described in the right column into groups that can be managed either by providers or consumers. For example, a provider like Digital Ocean provides a managed Kubernetes service that manages the data plane, the control plane and the cluster infrastructure. Only the application and cluster operations layers are completely free to manage. A customer can usually configure all the abstractions to some extent, however, if full customizability of the solution is needed, a more custom solution should be considered. Different providers and services offer offloading of different layers of the cluster functionality and thus simplifying the deployment and management procedure. However, it is again a compromise with the customizability and the right compromise should be chosen for a particular use-case.

This thesis uses a Digital Ocean managed Kubernetes cluster that already manages data plane, control plane and cluster infrastructure and simple to use for a small-scale solution that does not need customizability. Alternatively, a completely custom (bare-metal) solution can also be implemented using tools like Fedora, where nothing is managed by the provider.

A Kubernetes cluster is the biggest entity in Kubernetes, as it provides all parts needed for a working Kubernetes environment. A high level architecture of a Kubernetes cluster is described in Figure 3.12. There are two different types of nodes in a Kubernetes cluster called master and worker nodes. Master node is usually only one and is responsible for the management and orchestration of the whole cluster, which can include multiple worker nodes. Worker nodes manage containers, communication between them and assign resources to scheduled
Every change in the Kubernetes cluster usually starts with a kubectl command that performs a REST request to the API server in the master node. The API server validates the requests and executes the commands. As a result, the state of the cluster is saved in the etcd storage part of the master node, which acts as a distributed and consistent key-value store for the purpose of saving shared configurations and service discovery. For example, etcd stores details about scheduled jobs, pods, services and replication information. Deployments of pods and services depend on the scheduler component. Scheduler is aware of the available and required resources and can decide where to deploy scheduled services. For example, if a service/pod needs at least 2 CPU cores and 4GB of RAM, the scheduler deploys that service to only the node with the required resources. The last very important component in the master node is a controller manager that runs different controllers used in the Kubernetes cluster. Controllers use information provided by API server to observer the state of different resources and the changes in configuration. If there is a miss-match between the two, controllers trigger events to synchronize the configuration and the real state of the resources.
For example, Replication controllers are responsible for keeping a desired number of replicated pods by adding new ones or removing redundant ones. Node Controllers notice when a pod that should be running fails and recreate it, Endpoint controllers create end points to connect pods and services, Service account and token controllers create default accounts and API access tokens.

API server acts as a hub of all of the communication in the master node and communicates with worker nodes using the kubelet component that is included in each worker node. Kubelet is responsible for keeping the connection between the master and worker node. It ensures the required containers are running according to the configuration provided by the API server. Kubelet also communicates with the etcd to get information about services/pods and send information about newly created ones. Kube proxy is another component included in each worker node and acts as a network proxy by taking care of network routing for TCP and UDP packets.

Probably the most important part of the worker node is a pod. This is the smallest deployable entity in the Kubernetes cluster. Pods can contain multiple containers and containers in each pod share storage, namespace and IP addresses. Pods have a short life-time and should be easily rebuildable and replaceable in the case of failure or replication. Pods contain services and containers that are tightly coupled and would usually run on the same machine if Kubernetes was not used. All services and containers that should be decoupled, should run in a different pod. However, these principles are more important for a microservices architecture and are not discussed in detail.

However, pods usually require an additional layer of abstraction for deployment purposes and that layer is called the deployment. The deployment is a type of a controller that takes care of replica sets and the deployment of pods. Pods are deployed by a deployment controller that sets the number of replicas, the types of pods used and additional parameters for running pods. It is recommended to use deployments to deploy pods, but not necessary and there are
multiple ways of deploying pods and configurations that can be implemented for
a specific use-case.

Services are another important component in the Kubernetes cluster, as they
provide a gateway for multiple pods. The rebuildable and short life-time nature
of pods demands a more constant component that acts as a gateway to the very
volatile pods. Therefore, any entity that needs to communicate with a certain
pod, does not have to keep track if that pod is up or if it was replaced. Only
the details of the service, acting as a gateway, are needed to access the pods
functionality.

Services and deployments can already define a Kubernetes cluster running an
application. Both are usually defined in a single file that is named a Kubernetes
manifest. To upload a new configuration of the Kubernetes cluster, a manifest
is applied to the select cluster using the kubectl command which triggers events
in the master node that propagates instructions to the worker nodes through the
described architecture of components.

3.2.3 Delivery automation

This section describes the whole process of development of the deployment pipe-
line using GitLab Runners to deploy the Docker image that includes the code and
tools to run on the server node in a Digital Ocean managed Kubernetes cluster.
Requirements of this architecture are that the pipeline have as little dependen-
cies as possible and runs on a system that has only Docker installed on top of
Ubuntu (in this case GitLab runners are used). This is achieved by developing
Docker containers that use images that have the dependencies already installed
or a container installs them in the Dockerfile on top of the image. This way of
development brings positive characteristics as usability and portability, however
it also brings challenges, as most of the documentation provided does not predict
running everything in Docker containers. Most of the challenges that were enco-
untered are presented along with the solutions which probably provide the most
value to the reader.

First implementation of this deployment pipeline is with a manual part and full automation is not reached. At the time first implementation, credentials for applying the final production image to the cluster was only possible manually using Digital Ocean console. And a weekly rotation of the credentials by Digital Ocean for security purposes disables the possibility of a completely automated deployment process as the credentials still needed to be downloaded manually every 7 days to the deployment runner. In the second part of this section, a solution of this problem is presented as at a later date, Digital Ocean provided a tool for downloading the credentials.

The deployment pipeline described in this section demands two types of GitLab runners for execution of jobs. First runner is using a docker:simple image with a docker tag and the other is using a basic alpine:latest image with a python tag. The tags are crucial for running jobs on the right infrastructure, as the jobs with a docker tag will not work on a python runner.

![Figure 3.13: Workflow of the first version of the GitLab-Kubernetes deployment pipeline.](image)

Figure 3.13 presents a workflow that is implemented first for a semi-automated
deployment pipeline. The pipeline is triggered by the push to a production git branch on GitLab. The deployment pipeline is implemented with GitLab stages and jobs as described in Section 3.1.2. The first block presents a build of the Dockerfile located in the root of the project that defines a Docker image which contains the code of the application and the required dependencies. When this Dockerfile was developed running the python manage.py was considered:

```bash
$ python manage.py runserver
```

However, even if this is maybe the simplest solution and perfectly good for a local testing setup, it is not suitable for production. Instead of that option a Gunicorn (a python WSGI HTTP server for UNIX) is used, which is more appropriate for production and used by the Google App Engine platform. As a result, the following Dockerfile is implemented after some iterations:

```bash
FROM gcr.io/google_appengine/python
RUN virtualenv /env --p python3.5
ENV PATH /env/bin:$PATH
RUN apt-get update && apt-get install -y locales locales-all
ADD requirements.txt /app/requirements.txt
RUN /env/bin/pip install -r /app/requirements.txt
ADD . /app
CMD gunicorn -b :80 zeleniplanet.wsgi
```

The first line of code imports a Docker image that is publicly accessible and used by Google App Engine. All lines of code that come after the first line are unique for this container and add layers to the used image. In the next few lines a python virtual environment is activated and a PATH system variable is set for the bash shell to have the right scope. The fourth line of code is important, as it installs the packages for the local time, as that is used in the code. Here it is again apparent that the exact implementation of the architecture heavily depends on the code and use-case. The fifth line adds a requirements.txt to the storage of the Docker container. Always when an ADD command is used, the container image
Implementation of the architectures

grows in size as adds more storage to the scope of the container. Sixth line runs the
installation of python dependencies specified in requirements.txt. Seventh
line adds the whole root folder of the code and then in the next line specifies the
command that is run at the end when the container is run. The last command is
the most important one as it starts the Gunicorn server framework, which is the
whole purpose of the imported image in the beginning of the Dockerfile.

The described Dockerfile is built and uploaded to the image registry in the
GitLab job specified in the .gitlab-ci.yml file that represents the first two
blocks of the workflow diagram described in Figure 3.13.

Listing 3.6: A docker build job specified in the .gitlab-ci.yml

```
1 docker image build job:
2 stage: image_build_push
3 tags:
4  - docker
5 script:
6    - echo "Pushing the docker image"
7    - docker login registry.gitlab.com -u $DOCKER_REGISTRY_USER -p $DOCKER_REGISTRY_PASSWORD
8    - docker build -t registry.gitlab.com/primozkocevar/zeleniplanet/gunicorn:$CI_COMMIT_SHORT_SHA .
9    - docker push registry.gitlab.com/primozkocevar/zeleniplanet/gunicorn:$CI_COMMIT_SHORT_SHA
10 only:
11  - production
```

In the definition of the job, tag is used to specify that the job needs to be
run on top of a Docker runner. The only property is set to production so that
the job is triggered only when a change is pushed to the production branch. Two
environment variables docker registry password and docker registry user are used
to provide credentials for logging into a GitLab registry that saves built images.
Environment variables are set in the GitLab web console only by maintainers of
the projects and usually contain the most important secrets for the deployment
pipeline and the project, as they should not be included in the code directly. A docker build command builds an image from the described Dockerfile and specifies the name that should be like the location, where it is saved in the registry. At the end of the name, as a tag, a short hash of a git commit is appended, so that every image can be immediately connected to the specific code commit. The last command pushes the built image to the registry, so that it can be later used by the Kubernetes cluster if the proper credentials are provided, as this is not a public image.

As the architecture is using a gunicorn server, the static files should be served separately. This is presented with a block in Figure 3.13 and can be described in a GitLab job as with building a Docker image.

Listing 3.7: A static push job specified in the .gitlab-ci.yml

```plaintext
static push job:
stage: static_push
tags:
  - python
script:
  - echo "Pushing_the_static_files"
  - mkdir ~/.aws/
  - touch ~/.aws/credentials
  - pip install awscli
  - printf "[eb-cli]\naws_access_key_id=%s\naws_secret_access_key=%s\n" "$AWS_ACCESS_KEY_ID" "$AWS_SECRET_ACCESS_KEY"
  - aws s3 sync static/ s3://zeleniplanet-static-s3-bucket/static/
    --acl public-read
only:
  - production
```

This job is again defined by an only attribute and limited to the production branch. However, the tag is now different and set to python, as this job is using python packages to install awscli, which is a tool for managing
Implementation of the architectures

AWS services using the Command Line Interface (CLI). Again environment variables are used as a way of saving credentials, which are copied to a hidden file ./aws/credentials. This provides credentials for the last command of syncing the development static files with the ones served by the AWS S3 static file serving service in production.

In the place of the AWS S3, Digital Ocean Spaces were used at first and were quite simple to implement. However, the DO Spaces proved unreliable and slow for the use-case of serving different static files at that time. The implementation of Spaces should maybe be revisited in the future, as they will probably improve, but for now AWS S3 is a more common and reliable solution for serving static files.

![Figure 3.14: Workflow of the second version of the GitLab-Kubernetes deployment pipeline.](image)

The biggest issue of the first pipeline depicted in Figure 3.13 is the need for manual intervention of the system administrator to download the configuration file that contains the needed credentials to run the kubectl commands for...
applying new images. Even if the act of applying the image to a Kubernetes cluster could be automated, there is a need to download the changed credentials every 7 days and that breaks the automation in the pipeline. Applying migrations to a production database is also manual in the first pipeline.

After the first deployment pipeline was running in production for a few months, there was an update from Digital Ocean that provides a tool for downloading credentials automatically. This provided an opportunity to completely automate the deployment pipeline and update it. The final version of the deployment pipeline is shown in Figure 3.14. It is apparent that it is completely automated, and the events are only triggered by the commits to the git branches and after that no human interaction is necessary. There is still a challenge of running all jobs on runners with minimal dependencies (only python and docker) so that jobs are as independent of the underlying infrastructure and can be used with GitLab paid runners in the case of emergency. Therefore, all other dependencies are specified in the Dockerfile and installed in the container image or with bash commands in the case of python runners.

For the purpose of the automated download of credentials and the usage of these credentials to apply a new image to a Kubernetes cluster in an automated manner, a new GitLab job is developed. The Kubernetes apply job uses a `sed` command to replace a container image tag in the Kubernetes configuration file with a commit hash saved in the environment variable. This is necessary, as the image is built and uploaded by appending it with that tag in the build stage.
Listing 3.8: A GitLab job for applying a new container image to a cluster specified in the .gitlab-ci.yml

```
kubectl apply job:
  stage: deploy
  tags:
    - docker
  script:
    - echo "Building_doctl_and_kubectl_in_one_image"
    - sed -i -e 's/HASH_TAG/"$CI_COMMIT_SHORT_SHA"/g' cd/
      kubernetes/kubernetes_deployment.yaml
    - docker build --tag=kubectl cd/kubernetes/.
    - docker run --interactive kubectl apply -f
      kubernetes_deployment.yaml
    - docker run --interactive kubectl get pods
  only:
    - production
```

The core functionality of this job is important as it contains important parts of the deployment pipeline. The used Kubernetes deployment configuration file defines the way the container image is applied to a cluster and the orchestration properties of the cluster, which are discussed in the next Section 3.2.4. To use the Kubernetes configuration file, `kubectl` should be installed and provided with the correct credentials. This is both implemented in a single Dockerfile located in `cd/kubernetes/` folder.

At the moment of writing this master thesis `doctl`, the tool provided by Digital Ocean to control their resources from the command line, still does not implement a way to apply new images to Kubernetes clusters, but only enables the download of credentials. This is not too surprising as this is quite a new tool and in the future, it will probably be implemented. Hence for applying new images to a cluster, `kubectl` is used. A challenge is encountered here as credentials provided by `doctl` should be connected to the credentials used by `kubectl`, which are usually located in the `/kube/config` file.
When firstly implementing, two different Dockerfiles were considered, one for getting credentials with `doctl` and the other to apply the image using `kubectl`. A Docker volume was used in a testing implementation for the exchange of credentials between the two containers. However, that solution was complicated and did not work properly as the credentials downloaded by `doctl` were saved in `/root/.kube/config` file that was difficult to access from another container. The final solution was aggregated to a single Dockerfile, using both `doctl` and `kubectl`.

Listing 3.9: Dockerfile for downloading credentials and applying a new image

```
FROM alpine:3.8
ENV DOCTL_VERSION=1.20.1
RUN apk add --no-cache curl
RUN mkdir /lib64 && ln -s /lib/libc.musl-x86_64.so.1 /lib64/ld-linux-x86-64.so.2
WORKDIR /app
ADD . /app
RUN curl -L https://github.com/digitalocean/doctl/releases/download/v${DOCTL_VERSION}/doctl-${DOCTL_VERSION}-linux-amd64.tar.gz | tar xz
ENV DIGITALOCEAN_ACCESS_TOKEN="<your_token>"
RUN echo "$(./doctl kubernetes cluster kubeconfig show zeleni-planet -production01)" > credentials_config_kubernetes
# second part installing + using kubectl
ADD https://storage.googleapis.com/kubernetes-release/release/v1.6.4/bin/linux/amd64/kubectl /usr/local/bin/kubectl
ENV HOME=/config
RUN set -x && 
    apk add --no-cache curl ca-certificates && 
    chmod +x /usr/local/bin/kubectl && 
ENV KUBECONFIG=credentials_config_kubernetes
ENTRYPOINT ["/usr/local/bin/kubectl"]
```

This Dockerfile uses a classic alpine Docker image and installs `doctl` on top that is used for downloading the credentials that are copied to a config
Implementation of the architectures

file credentials_config_kubernetes. In second part of the Dockerfile, kubectl is installed and used with credentials saved in the config file. At the end a configured kubectl command is provided as an executable ENTRYPOINT so that any operation using kubectl can be initiated using this container. In this job the container is used for applying a new image and checking the state of the pods after:

- docker run -i kubectl apply -f kubernetes_deployment.yaml
- docker run -i kubectl get pods

Another upgrade in the final pipeline is the usage of stages to group jobs that can be run parallely. Uploading static files and applying a new image to a Kubernetes cluster can be done at the same time as they are independent from each other. However, they both depend on the result of building and pushing a docker image. That is why building and pushing a docker image is in the first stage named build, which is reflected in the stage attribute in the .gitlab-ci.yml file. If the build stage is successful, the deploy stage can start. The deploy stage parallely runs both jobs of applying the new image to a Kubernetes cluster and uploading static files to AWS S3. This is reflected in the Figure 3.14 by dotted lines that separate different stages.

Listing 3.10: Jobs grouped into stages in the .gitlab-ci.yml

```yaml
stages:
- build
- deploy
docker image build job:
  stage: build
...
kubernetes apply job:
  stage: deploy
...
static push job:
  stage: deploy
...```
3.2.4 Digital Ocean and Kubernetes configuration

Deploying required components for the Digital Ocean architecture presented in the second block in Figure 3.7 is relatively simple using the Digital Ocean web console. Lately there were some developments of a CLI tool doctl, however at the time of writing this thesis, doctl is a very new tool and a lot of features are unsupported.

Initialization of a managed Kubernetes cluster is done on top of a DO droplet using 2GB of RAM and 50GB of storage, as this is a minimum requirement for running a Kubernetes cluster. Monitoring of the cluster is done using the web-console and for more advanced configuration of the Kubernetes cluster kubectl is used. If manually running kubectl and configuring the Kubernetes cluster for testing or debugging purposes, kubectl with docker should be installed on a development machine first. Additionally, proper credentials and configuration should be supplied to a /kube/config file (if the development machine is running Ubuntu), downloaded from a DO web interface. When everything is set up correctly the kubectl can be used for testing or debugging manually, but is not recommended in production, as the infrastructure should be as immutable as possible.

A similar service to AWS Route 53 is provided in the form of Domains. Domains service is used to set different DNS records that redirect requests according to the configuration. Mail server records are handled with MX records and classic IPv4 addresses are redirected using A type records. Table 3.2 provides a working configuration of DNS records on DO for a domain running in production. It is a simple setup, redirecting from the main domain and the platform subdomain to the platform server located on 139.59.204.196. Mail server is located on a different platform, using an 195.246.8.165 IP and two DNS records are needed to implement that forwarding (first and second). NS records are used to transfer the control of DNS settings from the owner of the domain to DO by using the NS records. If the domain is transferred to DO, certificates can also be managed as
Table 3.2: Configured DNS settings in DO web interface using Domains (domain.com is used as an example). DO sets TTL values by default.

<table>
<thead>
<tr>
<th>Type</th>
<th>Hostname</th>
<th>Value</th>
<th>TTL (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MX</td>
<td>domain.com</td>
<td>mail.domain.com</td>
<td>14400</td>
</tr>
<tr>
<td>A</td>
<td>mail.domain.com</td>
<td>195.246.8.165</td>
<td>3600</td>
</tr>
<tr>
<td>A</td>
<td>platform.domain.com</td>
<td>139.59.204.196</td>
<td>3600</td>
</tr>
<tr>
<td>A</td>
<td>domain.com</td>
<td>139.59.204.196</td>
<td>3600</td>
</tr>
<tr>
<td>A</td>
<td><a href="http://www.domain.com">www.domain.com</a></td>
<td>139.59.204.196</td>
<td>3600</td>
</tr>
<tr>
<td>NS</td>
<td>domain.com</td>
<td>ns1.digitalocean.com</td>
<td>1800</td>
</tr>
<tr>
<td>NS</td>
<td>domain.com</td>
<td>ns2.digitalocean.com</td>
<td>1800</td>
</tr>
<tr>
<td>NS</td>
<td>domain.com</td>
<td>ns3.digitalocean.com</td>
<td>1800</td>
</tr>
</tbody>
</table>

in the case of AWS certificate manager. DO uses Let’s Encrypt open certificate authority to provide free SSL certificates and automatically renews them when necessary.

The IP address 139.59.204.196, used as a destination in forwarding from the base domain, www. and platform. subdomains is the IP address of a load balancer that forwards the request to the Kubernetes cluster. The load balancer is defined as a Kubernetes service in a Kubernetes manifest, where exposed ports, names and tags are defined. Details of the implementation are available in the file kubernetes_deployment.yml.

Managed database is a new service provided by DO and is used for database support in this architecture. It is easy to set up and runs with 1GB RAM, 10GB of disk storage and 1 virtualized CPU, which can be scaled up easily if needed. It is again based on a PostgreSQL database engine. The managed database provides built-in backups and recovery with monitoring and logging. Insights provided by the web console can be very useful to analyze what is happening with the database and react on the events. Custom monitoring and alert properties are set accordingly (e.g. if CPU is above 80% for 10 minutes send an alert).
Securing the database is done by only allowing specific trusted sources to access the database. The Kubernetes cluster is therefore added as a trusted source, so that it can connect to the managed database. There are also database credentials provided and are used in the Django production settings of a project, thereby the connection of the managed database and the Kubernetes cluster is successful.

Applying configuration to the Kubernetes based orchestration system is usually a much less graphical experience compared to configuring Elastic Beanstalk through the web interface. To configure and apply a Kubernetes deployment, kubectl with proper credentials is used. A deployment can be applied using kubectl by multiple commands to create, expose and authenticate resources. Doing this with multiple commands is mostly used for testing and debugging as this is not in line with the infrastructure as code principles.

Aggregating all the commands in the single file that is called a Kubernetes manifest, is more appropriate for production and implemented in a kubernetes_deployment.yaml file. This approach suits the immutable nature of the architecture much better. The approach used is adapted from the recommended implementation by the Google Cloud documentation [26].

The deployment manifest describes the implemented Kubernetes architecture and analyzing it provides a good insight in to the caveats and challenges. At the beginning of the definition of a Kubernetes object, an API version is defined as it describes the type of API server that is used to receive the kubectl commands specified in the manifest of the object. For example, extensions/v1beta1 API version is used when defining deployments as this is the version that worked best with the configuration. Alternatively, for the specification of a Service, v1 API is used. The version of the API depends on what features are used in the configuration of the object and if that features are supported by the used API version.
Listing 3.11: A kubernetes_deployment.yaml file defining a Kubernetes manifest.

```yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: zeleniplanet-django
  labels:
    app: zeleniplanet-django
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: zeleniplanet-django
    spec:
      containers:
      - name: zeleniplanet-django-app
        image: registry.gitlab.com/primozkocevar/zeleniplanet/
gunicorn:HASH_TAG
        imagePullPolicy: Always
        ports:
        - containerPort: 80
        imagePullSecrets:
        - name: gitlab-auth
---
apiVersion: v1
kind: Service
metadata:
  name: zeleniplanet-django
  labels:
    app: zeleniplanet-django
spec:
  type: LoadBalancer
  ports:
  - port: 80
    targetPort: 80
  selector:
    app: zeleniplanet-django
```
Secondly, a type of the Kubernetes object is chosen using the kind tag. Manifest that is defined, uses two different types of objects. It defines one Deployment and one Service, which is a simple set-up, enough for the needs of this thesis. There are a lot of different types of objects available for definition in the Kubernetes manifest file and similar results can be achieved by using different approaches. For example, instead of using a Deployment, pods could be defined manually or by using Replica sets, which would result in a more complex manifest file that would implement a similar Kubernetes architecture.

Each object also needs some metadata to provide context such as name and labels. Labels are used for connecting different objects like Services and Deployments. Service uses the selector that matches the label of the Deployment. That connects the Service and the Deployment and directs traffic from the Service to the Deployment.

Specification field defines the content of the object. In the case of the Service, a type of service is defined as a load balancer, as that is the function of this service. Ports are also defined in the specification. Traffic to the port 80 on the service is directed to port 80 on the underlying pod. This could also be customized for a more complex setup or if a server is run on a different port.

The specification of the Deployment is a bit more complex as it must provide more context about the application. The number of replicas are specified in the beginning, which creates a replica set controller that guarantees a set number of pods are always running. Here it is set to one, as running more replicas would be too resource consuming for the small scale of the application. However, if scalability can quickly be achieved by only changing the number of replicas without the need of buying new hardware or installing additional software, as is the case with classical bare metal configurations. Providing a docker image for the container that is about to be deployed is the next big step. If the image is served by a public registry, the procedure is quite simple as the URL of the image is supplied and applied. However, when using a private registry, for serving
private docker images, credentials need to be provided first, for the pull of the image to work. A private registry on GitLab is provided with the repository and is therefore used. Connecting the Kubernetes cluster and a GitLab private registry proved challenging at first. The best solution is implemented in a form of a Kubernetes secret that is created by the kubectl tool.

```
kubectl create secret docker-registry gitlab-auth --docker-server=<your-registry-server> --docker-username=<your-name> --docker-password=<your-pword> --docker-email=<your-email>
```

After this command is run with the proper credentials, the selected Kubernetes cluster has the right authorization to pull an image from an authorized private registry. The secret that is used for authorization is specified as an imagePullSecret. Exposed ports on the container are specified and should match the exposed ports in the Dockerfile and directed ports in the Service for the internet traffic to be correctly directed. Image pull policy is set to always as that proved the most reliable, even if it can be inefficient sometimes. The always setting ensures always the most recent container image is fully uploaded and no conflicts with previous images are possible.

The complete kubernetes manifest defined is applied using kubectl apply command as specified in the previous chapter. Described mechanisms in this Section update the configuration accordingly and start with orchestration processes to synchronize the desired configuration and the state of the cluster. During the lifetime of the cluster, described mechanism also provide replication, healing, monitoring, management and other capabilities that add to the quality of the application.
4 Results

This section describes the results of the implemented architectures using different metrics. For example, the resulting availability of the solution depends on the implemented architecture, however it depends on the actual coding implementation of the platform as well. As the coding implementation of the solution is out of scope of this thesis, these kinds of metrics that are very dependent on the actual coding implementation are omitted from the results.

The metrics are divided between the results of the deployment pipelines implemented and results of the implemented cloud architectures. The results act as a proof of the implemented architecture and a comparison point to the more manual architectures. After the analysis of the results, appropriate conclusions are made and connected with the goals set in the Chapter 2.

4.1 Deployment pipeline metrics

Deployment pipeline for the implemented architectures was implemented gradually through the development of the project. At first only the cloud architectures used for serving the solution were set up. The deployment at this stage was manual and was quite complex as it demanded a lot of dependencies on a specific machine that had all the appropriate credentials and only a single person that knew the procedure could deploy from this machine. This presented both a scalability and a security bottleneck in the whole architecture as it depended on a single machine and a single person.
As the project was developed further, an opportunity for an automated solution was presented. Both AWS based and Kubernetes based architectures already included tools that made the automation of the deployment procedure possible. However, the Kubernetes-based architecture demanded more work as it is fundamentally more complex and did not provide a way of getting the credentials automatically at the beginning, as described in Section 3.2.3.

The AWS-based architecture was also simpler to deploy manually as it did not have a lot of deployment dependencies other than GitLab project credentials, awscli tools and AWS credentials. When this was properly set up on the machine the person deploying usually needed 5-10 minutes to get the new code, merge the branches, do basic checks, deploy and turn on the worker. This measurement is approximated from the experience of running more than 200 manual deploys during the time span of approximately 10 months, in the time of May 2018 to March 2019. Experiences and measurements for this assessment were gathered on multiple different projects based on the AWS architecture as well as the web scraper described in this master thesis.

The development of the automated pipeline for the AWS project started on March 2019 as seen in Figure 4.1. It can be observed that at the beginning, the pipelines were failing more, as they were still under development. However, after
the pipelines were successfully developed, only successful jobs are observed. The decline in the number of pipelines in the time of May 2019 and July 2019 is only connected with the decreased activity of development of this project in that time and is not important for this thesis.

The result of the automated deployment pipeline is presented on Figure 4.2 where an example of a passed deployment pipeline is presented. The pipeline deploying to AWS is very simple and composed of only one deploy job that runs multiple consecutive tasks automatically, as described in Section 3.1.3. This job is run specifically on a GitLab runner with a tag python, which means it needs python installed. All of the mentioned automated tasks are finished in 1 minute and 24 seconds if everything is successful. Comparing this to the deployment time of 5-10 minutes measured when deploying manually, the time of approximately 1 minute and a half is a big improvement and a good compromise with portability. A comparison of durations for an automated pipeline and a manual pipeline is available in Table 4.1.

<table>
<thead>
<tr>
<th>Type</th>
<th>Duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manual</td>
<td>5 - 10 minutes</td>
</tr>
<tr>
<td>Automated</td>
<td>84 seconds</td>
</tr>
</tbody>
</table>

The Kubernetes-based deployment pipeline was much more complex to run completely manually, as it required the building of Docker images and pushing
them to the public repository. Downloading credentials was required manually and pushing of the static files to AWS S3 was needed. This was again only possible from the single machine that had access to credentials and a single person that had the proper knowledge of the whole procedure. At the beginning approximately 20 deployments were made completely manually that required on average 10-15 minutes from getting the credentials in the beginning to applying the updated image to the Kubernetes cluster at the end. This procedure was also quite complex and was not appropriate for manual work.

Consequently, a partly automated deployment pipeline was implemented, where the building and pushing of the container image to the registry was automated in a single GitLab runner job named image build push. Syncing the static files to AWS S3 was also automated in a separate job named static push. The results of the automated procedure are presented in Figure 4.3 where each job usually took around 40 seconds. At this stage, the whole deployment process usually took 5-10 minutes as the automated procedure first had to be triggered and after it was finished, the configuration had to be applied manually to the Kubernetes cluster. For all of this to work, proper credentials had to be downloaded every week to the deployment machine.

After the Digital Ocean tool doctl enabled the downloading of credentials automatically, the whole deployment process was automated. The results of this deployment pipeline are presented on Figure 4.4 where the pipeline is separated in a build and a deploy stage. Build stage builds a docker image and pushes it to
4.1 Deployment pipeline metrics

Figure 4.4: The results of the implemented deployment pipeline in GitLab for deploying to Kubernetes.

the registry and takes approximately 40 seconds. If the build stage is successful, the deployment stage triggers two parallel jobs. The job that applies the new image to Kubernetes and downloads the proper credentials, takes approximately 40 seconds. The duration of the static push job depends on the amount of files that need to be pushed however, generally it takes less than a minute. Summing the times together, the automated deployment procedure takes 1 minute and 26 seconds in this case and generally a minute and a half. Comparing this to a time of 10-15 minutes for a wholly manual procedure and 5-10 minutes for the partially automated one, a big improvement in deployment time can be observed. The duration comparison of automated, semi-automated and manual deployment pipelines are available in Table 4.2.

Table 4.2: Kubernetes deployment pipeline duration comparison

<table>
<thead>
<tr>
<th>Type</th>
<th>Duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manual</td>
<td>10 - 15 minutes</td>
</tr>
<tr>
<td>Semi-automated</td>
<td>5 - 10 minutes</td>
</tr>
<tr>
<td>Automated</td>
<td>86 seconds</td>
</tr>
</tbody>
</table>
The deployment pipeline time in both cases could also be shortened by providing a more powerful runner or pre-installing more dependencies, but that would also negatively affect portability as the runner would have to be more specific. Making the deployment faster and sacrificing portability would not make any sense for this use-case, however could be useful if deploying every minute is necessary. More importantly, with the automated deployment pipeline, the deployment process can be triggered from any machine/person that can push to a selected branch and is not limited as it was with the manual process.

### 4.2 Cloud system snapshots

This section of results provides metrics snapshots of resources that are running the implemented architecture. Provided snapshot give an insight into how the architecture is running and utilizing the resources, which provides an insight into efficiency and scalability.

![CPU utilization](image)

Figure 4.5: CPU utilization in percentage of processing power being used across all cores for a droplet running in a Kubernetes cluster.

Figure 4.5 presents CPU usage of a Digital Ocean computing instance that acts as a Kubernetes node. The provisioned CPU uses only 1 virtual CPU core provided by Digital Ocean. It can be observed that CPU utilization never reached values higher than 60% for a longer period and is usually running under 30%. Occasional peaks in the CPU usage that are quite periodical appear because
of the scraper worker that is usually triggered every day on this instance. The scraper worker usually runs for 3 hours and scrapes different websites for content. However, in the beginning of July, the scraper was turned off and that can be observed on the graph as the peaks are not present. Besides the peaks, load varies based on incoming traffic, which is a quite stochastic process but does not affect CPU utilization too much. This kind of results are a good compromise between under and over-provisioning for our use-case and means resources are appropriately provisioned.

![Load average](image)

**Figure 4.6:** Load average for a droplet running in a Kubernetes cluster. The Droplet uses 1 vCPU.

Load average presented on Figure 4.6 gives even more insight into resource utilization. Linux load average is a metric based on the number of CPU cores used and explains how many tasks are waiting/using the resources, normalized by the number of used cores and averaged over 5, 10 or 15 minutes. As the number of cores in the measured system is only 1, the metric is quite simple and generally it is important to keep the load averaged over 15 minutes, presented as a dark blue line, under the value of 2. Peaks in the load averaged over 5 or 10 minutes are not so crucial for our use case. Therefore, it can be observed that peaks in the 5 minute and 10 minute curve occur, but are only temporary and should not be problematic. They are probably caused by the scraper worker as they again appear periodically.

Memory utilization on the Digital Ocean computing instance is presented in
Results

Figure 4.7: RAM usage in percentage of all available RAM for a droplet running in a Kubernetes cluster. The measured droplet uses 2GB of RAM.

Figure 4.7. The provisioned RAM is 2GB and thus 100% of utilization would mean all 2GB of memory are used. Memory utilization is almost constant at around 50%, which is again a sign of good provisioning for this use-case.

Figure 4.8: Public internet bandwidth over the public internet interface that connects the Kubernetes cluster to the internet.

Network throughput is a metric that depends the most on the usage of the platform and is presented in Figure 4.8. Outgoing and incoming traffic patterns are matched quite well, where outgoing traffic is usually a bit higher. The scraper worker also affects the network throughput periodically when it is active. No special anomalies are detected observing this graph.

The second part of this section describes the snapshots of the AWS resources used in the AWS architecture. More specifically, the metrics provided measure
AWS computing instance named EC2. In this case, a t2.micro EC2 instance with 1 virtual CPU core and 1GB of RAM is used.

Provisioned CPU on a used EC2 computing instance uses only 1 virtual CPU core and the utilization of that is presented on Figure 4.9. Periodical peaks are present each day when the scraper worker is turned on and CPU utilization raises above 30% consistently. When the worker is not active, the load is very low as this architecture was used only for testing and was not in production as compared to the Kubernetes setup. Therefore, there was not a lot of actual traffic that would use a lot of resources. However, the whole graph is quite consistent, which is a good sign. If the utilization was consistently high, more resources would be automatically provisioned.

4.3 Platform neutrality

Platform neutrality and interoperability of the technologies used is very important, when looking at the alternatives when choosing an architecture appropriate
for a use-case, which is already explained in the Chapter 2.

Two presented solutions have very different levels of platform neutrality. The AWS based architecture is much less interoperable than the Kubernetes based one. The implemented AWS architecture uses a lot of tools such as Elastic Beanstalk, Route 53, RDS... All those tools are mostly proprietary and developed only by AWS and can thus only be used with AWS. This presents a decent amount of vendor lock-in and reduces flexibility and portability of the architectures. For example, if AWS increases the price for the Elastic Beanstalk service or cancels it, the implemented solution would be forced to adapt, which would require a lot of resources. This fact proves the lack of portability and flexibility to the external changes, making the feedback loop of responding to provider changes longer.

Alternatively, the Kubernetes based solution provides similar functionalities while sacrificing simplicity to achieve a more platform neutral architecture. Resource orchestration provided by Kubernetes can be used in multiple different platforms and providers such as AWS, Azure, Google Cloud, Digital Ocean... Open source nature of Kubernetes as an orchestration tool provides an ability for it to be used with physical infrastructure and thus provides even more flexibility as it can be run completely independent of any providers if that is necessary. The described Kubernetes-based architecture could therefore be ported to a different provider much easier compared to the AWS-based one. There is even an option to implement all the technologies used in this architecture (e.g. Docker, GitLab, Kubernetes) on top of a completely physical infrastructure that would decouple the solution from any providers.

4.4 Comparison

This section describes a comparison of three different architectures discussed in this thesis. The comparison is based on subjective and objective parameters. For example, ease of use or portability is difficult to assess completely objectively
without a big survey, as they depend on a specific use-case and a specific developer. Therefore, the results presented in Table 4.3 are aggregated from metrics and experiences observed during the implementation and analysis of two different architectures described in this thesis. The classic physical server architecture is not described in this thesis but is still used for comparison.

Table 4.3: Comparison of different architectures for web platform hosting, assessed on the scale from 0-5.

<table>
<thead>
<tr>
<th>QoS</th>
<th>AWS based</th>
<th>Kubernetes based</th>
<th>Physical server</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalability</td>
<td>5</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Ease of use</td>
<td>5</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>Resiliance</td>
<td>4</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Portability</td>
<td>3</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Platform neutrality</td>
<td>1</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>SUM</td>
<td>18</td>
<td>18</td>
<td>11</td>
</tr>
</tbody>
</table>

Both AWS and Kubernetes based cloud architectures are able to scale, as their resources can be provisioned easily. AWS provisions resources automatically when needed, which is the best. The Kubernetes-based architecture is hosted on DO, which still does not support auto-scaling and therefore gets a lower score of 3 in this category. A classic physical server architecture is difficult to scale as it requires new hardware and a lot of setting-up.

The most simple to set-up and use with no prior knowledge is the AWS EB-based architecture, as it provides a simple way of starting an EB instance. Kubernetes-based architecture is fairly simple to set up, but is more complicated to start using, as it requires a better understanding of how Kubernetes works. A physical server can be simple to set up for experienced system administrators, but can be complicated for inexperienced developers.

Resilience is high in both cloud-based architectures, as they both implement mechanism for monitoring, alerting, logging and replication in the case of failure.
However, none of these are implemented in a basic physical server architecture.

Portability is the highest for the Kubernetes-based architecture as it can be installed on physical infrastructure, as well as it can be used with most cloud providers, therefore changing the underlying infrastructure is not difficult. AWS is easy to move inside the AWS cloud system, but cannot be used outside. Physical architectures are usually difficult to move to a different infrastructure as they require the most dependencies and can be bound to specific hardware and OS.

In platform neutrality, physical architectures are the best, as they are completely decoupled from cloud providers. Kubernetes is a good alternative as it can be used with most providers and on physical hardware, but is still an open-source tool developed by Google. EB is not able to work outside of AWS and presents vendor lock-in.

Looking at the final score in Table 4.3, both cloud architectures have the same score. However, as the Kubernetes was used with DO, it still did not implement some auto-replication functionalities that are already available using Google Cloud. Therefore, the benefits of the Kubernetes-based architecture will probably grow in the future and coupled with better portability and neutrality it presents a better choice for a long term solution, compared to the AWS EB based architecture.

Keeping in mind, the provided scores are based on the web scraper use-case, which defines a specific perspective. In a different use-case the results could be different. For example, the physical architecture, combined with the right tools, can sometimes provide similar functionalities to the cloud alternatives and can prove a viable option for some use-cases.
5 Conclusion

Using open-source tools provided by GitLab and Docker in the deployment process provides a very portable and fast way to deploy new code to the cloud architecture. This enables more user-focused and developer-friendly software implementation. The deployment architecture is also portable and can be run on a physical computer or any kind of cloud computing instances (e.g. Droplet, EC2...).

When comparing two presented architectures, AWS-based one enables easier use of desired functions such as replication, auto-scaling, backups, alarms... However, this comes at a cost of interoperability and portability as the architecture only works with AWS and can not be used with any other provider. The Kubernetes-based architecture is more interoperable as multiple providers use Kubernetes for orchestration purposes. This also means more developers know how to use the Kubernetes architecture as it is widely used. However, some basic functions that are already implemented in AWS, require some knowledge to implement with Kubernetes. Additionally, the Digital Ocean managed Kubernetes instance that is used in this thesis is quite new and at this time still does not support features like auto-scaling. If combining Kubernetes with a Cloud provider for a bigger project, Google Kubernetes Engine is recommended as a more expensive, but also more refined alternative.

However, both architectures provide multiple benefits when compared to a more classic physical server architectures as the deployment process proved to be much easier and faster. Additionally, during the life-cycle of the project, the
resources in the cloud-based architectures proved much more flexible when repli-
cating instances, scaling, getting monitoring data and adapt the infrastructure to
new circumstances. Setting up a similar type of cloud-based architecture when
starting with a project is recommended as it brings benefits in the long run as the
architecture does not need to be re-factored but only reconfigured if the project
grows.

5.1 Limitations

It is difficult to propagate all of the solutions and knowledge, obtained while
implementing two systems described, in a single master thesis. For more details
about implementation and the encountered challenges, other publications should
be considered [27] [28] [29].

The deployment pipeline integrated, provides support for an easy implement-
tion of automated testing and continuous integration if the application would
demand it. It must be mentioned that the implementation of automated testing
and continuous integration could be necessary in some use cases and continuous
delivery could be a dangerous tool if not used wisely. However, the application
described, does not demand the implementation of automated testing, therefore
automated testing is omitted as it would be out of scope.

5.2 Further research

In future, an update of this work with an addition of automated testing and
continuous integration, would be useful to consider as it would provide an even
more resilient and usable solution. For such an implementation a different use
case should be considered, one that demands more resilience and testing.

An additional architecture presenting an implementation on completely physi-
cal infrastructure would present an interesting non-cloud alternative to presen-
5.2 Further research

ted architectures. Technologies presented in the Kubernetes-based architecture are mostly already ready for on-premise implementation and only a few changes would be required to remove the whole architecture completely from the cloud to physical servers.

A more detailed and thorough analysis of objective attributes such as developer friendliness and user focus would be required to better measure the effects of the implemented architectures. Multiple different projects could be observed and interviews with developers and users could be obtained for more relevant results.
Literature


