



**Silesian
University
of Technology**

MASTER THESIS

„Selection and Analysis of tools used for automating deployment processes”

Mateusz KLIMAS

Student identification number 273697

Programme: Informatics

Specialization: Informatics Internet of Things

SUPERVISOR

Dr hab. Eng. Dariusz Mrozek, Prof. PS

Faculty of Automatic Control, Electronics, and Computer Science

GLIWICE 2022

Thesis title:

Selection and analysis of tools used for automating deployment processes

Abstract:

In the software development lifecycle repetitive tasks, done on a daily basis, should be automated in order to save time and avoid unnecessary issues. Building, testing, and deploying the application are examples of such processes. Designing an environment responsible for performing automation is a challenging task, due to its complexity and numerous technologies or platforms available on the market. This thesis highlights the importance of automation and its implementation in a software development lifecycle. It proposes an example automation system, along with the analysis and comparison of the available solutions.

Keywords:

Automation, deployment, CI/CD, cloud

Tytuł pracy:

Wybór i analiza narzędzi wykorzystywanych do automatyzacji procesów wdrożeniowych

Streszczenie:

W cyklu życia oprogramowania powtarzalne zadania, wykonywane na co dzień, powinny być zautomatyzowane, aby zaoszczędzić czas i uniknąć niepotrzebnych problemów. Budowanie, testowanie i wdrażanie aplikacji to przykłady takich procesów. Zaprojektowanie środowiska odpowiedzialnego za realizację automatyzacji jest trudnym zadaniem, ze względu na jego złożoność oraz liczne dostępne na rynku technologie czy platformy. Ta teza podkreśla znaczenie automatyzacji i jej implementacji w cyklu życia oprogramowania. Proponuje przykładowy system automatyzacji wraz z analizą i porównaniem dostępnych rozwiązań.

Słowa kluczowe:

Automatyzacja, wdrożenie, CI/CD, chmura

Index

1. Introduction	8
2. Theoretical background	9
2.1. DevOps Methodology	9
2.2. Version Control	11
2.3. Continuous Integration Continuous Delivery	15
2.4. Containerization	16
2.5. Container Orchestration	19
2.6. Public Cloud	23
3. Platform and Technologies	28
3.1. Application	28
3.2. Cloud Providers	31
3.3. Build Tools	39
3.4. Automation Servers	42
3.4.1. Jenkins	44
3.4.2. CircleCI	48
3.5. Summary	51
4. Project Implementation	54
4.1. Project Repositories	55
4.2. Cloud Environments	58
4.3. CI/CD Pipelines	62
4.4. Deployments	68
5. Experiments	72
5.1. Cost of the Resources	72
5.2. Regions and Latency	74
5.3. CI/CD Tools	75
5.4. Deployments	78
6. Summary	80
Bibliography	82
List of abbreviations and symbols	88
List of Figures	91
List of Tables	93

1. Introduction

Nowadays, in a modern, fast-growing world full of demanding end users, creating a good application is not enough. People, constantly, are expecting new features and improvements. If authors of the solution won't satisfy mentioned needs, users will likely shift to competition, stop using the product, or the application will not attract new users. Therefore, constantly improving every aspect of the solution is mandatory. Thus, the idea of continuous integration and continuous delivery (CI/CD) was born.

With such an idea in mind a lot of operations, like testing, building, and deploying applications, had to be done daily. Having lots of repetitive tasks, automation of these processes was introduced. The goal of this thesis is to highlight the importance of the automation of the processes and the importance of appropriate implementation of a system responsible for performing automated tasks.

In chapter 2 of the thesis, theoretical concepts behind the main technologies and methodologies used for this work are discussed. The next chapter is focused on choosing the most suitable platforms and technologies that will fulfill the assumed goals. Chosen will be things such as cloud providers, build tools, or automation servers, among others. Then, in chapter 4, based on the choices made in the previous section, automation and target environments are implemented. Besides that, the chapter contains all necessary actions and configuration steps with a detailed explanation. Chapter 5 includes all experiments performed to make the best possible choice for chapter 4. Apart from that, it also describes an experiment, which shows the benefits coming from the automated deployment process.

2. Theoretical background

The idea of automation of deployment processes is a very broad topic that involves many different ideas, practices, methodologies, and technologies. Thus, to not get lost, there is a need for distinguishing them. At a first glance, numerous technologies, can be confusing and a lot of questions might be risen, like: “what advantages come from given a methodology?”, “how do they all cooperate?”, “why is there a need for this technology”, and many more.

To overcome this obstacle and avoid unnecessary doubts in further chapters, this thesis starts by explaining key concepts and mechanisms that stay behind deployment processes and their automation. The chapter starts with an explanation of what DevOps methodology is and what practices it uses. It is followed by the introduction of very important concepts when it comes to automation, which are: Version Control, Continuous Integration, and Continuous Delivery, containerization, container orchestration, public cloud, and finally software deployments.

2.1. DevOps Methodology

Modern software development projects consist of numerous different modules and components, known as microservices. This strategy simplifies the development of an application, in a way that every service can be developed separately by a different team. In a system where each microservice is essential for correct execution, communication between people becomes essential. The information must be accurate, widely available, quickly found, and preferably supplied constantly, in real-time, to all team members to flourish in a world where technologies, needs, ideas, tools, and timeframes are always changing. Modern software development has evolved to include fundamental notions of flexibility to change to overcome these problems [1],[2].

Each project must start from an idea and specify its requirements. The Software Development Life Cycle (SDLC) is a conceptual model for planning, creating, testing, and delivering software [2]. There are three key milestones in the evolution of software development. The first step was to introduce the waterfall approach. The well-known paradigm, in which the process is viewed as a series of steps leading to project completion. This sequence describes how each stage's output becomes the following stage's input. Though simple to conceptualize and plan, when applied to the practice of software development, this technique is considered as having severe flaws, particularly the difficulty in reacting to change or new knowledge throughout the development process.

To overcome encountered problems with using the waterfall approach, the agile idea was proposed. Agile development methodologies are based on the fact that ever-changing requirements, constraints, and customer needs will continually impact all areas of a project throughout its life cycle and are designed to ensure an engineering team's ability to adapt to rapid and often significant change while maintaining project momentum [3]. Apart from its benefits, agile methodology is not flawless. Reduced cooperation as project deadlines lengthen, as well as the detrimental effects of incremental delivery on long-term outcomes, are two common difficulties found in agile development.

Having negative aspects of agile in mind, with the start of the year 2008 DevOps was proposed by developers Andrew Clay and Patrick Debois. The abbreviation “DevOps” comes from combining development and operations whose main task is to deliver continuous software improvements. By aligning and coordinating software development with IT operations, DevOps is a collection of adaptable strategies and processes that organizations employ to create and deliver applications and services. Development and operations teams can minimize bottlenecks by collaborating and focusing on improving how they produce, deploy, and continually monitor software by working together [4].

DevOps can be interpreted as an infinite cycle. Its most common visual interpretation is presented in Figure 2.1:

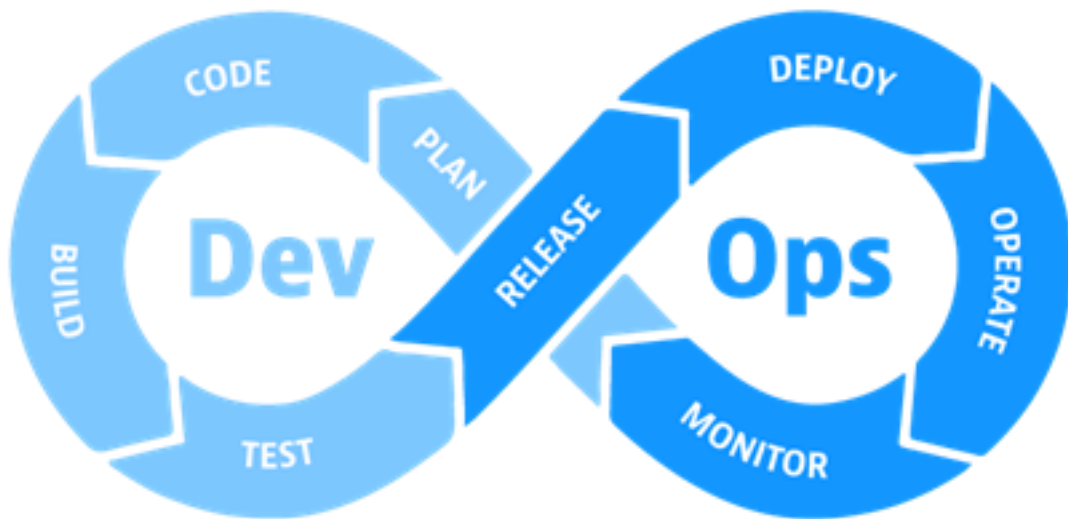


Figure 2.1 *DevOps methodology cycle*,
Source: <https://www.dynatrace.com/news/blog/what-is-devops/>

As shown in the figure, DevOps consists of eight main components that are executed continuously and often in parallel. Apart from the planning phase, every other can be fully or partially automated as will be shown in further chapters.

Enabling DevOps in a project causes many different benefits. Some of them are rapid delivery, reliability, and above all improved collaboration between operation and development teams. The ideas of agile development – rapid, bite-sized enhancements based on priority – may be applied across the full software life cycle by partnering or combining development and operations teams. The initial design, proof of concept, testing, deployment, and eventual revision are all included [4].

DevOps on its own couldn't accomplish much. It needs to be accompanied by different technologies and tools. As a starting point for it, as for other countless projects, or different methodologies, is version control. For this reason, the next chapter will be dedicated to describing this technology.

2.2. Version Control

A version control system (VCS) keeps track of changes made to a file or collection of files over time; thus, providing a history of changes. The most often use case is for storing and keeping track of changes for source code files, but VCS exceeds that. It allows to compare changes over time, check who introduced the change or the issue, and store the timeframe for it. Apart from that, it provides the possibility of rolling back the whole project or only selected files. First primitive VCS systems acted like a local database that stored all the changes made to the files, which is shown in Figure 2.2. The first improvement that came shortly after was the Centralized Version Control System (CVCS) which allowed developers to cooperate on a single project and then became a standard for many years [5]. The scheme of this System is presented in Figure 2.3.

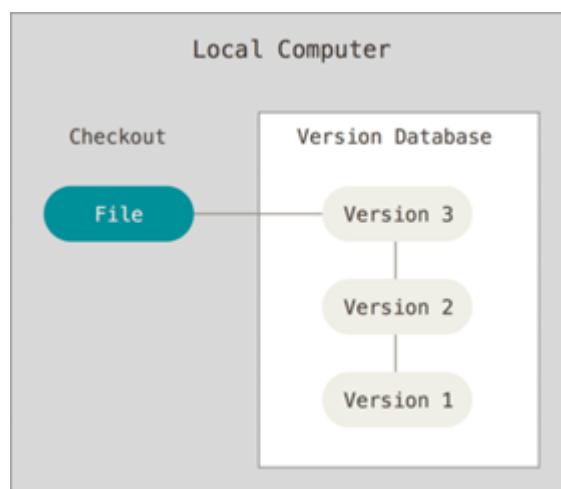


Figure 2.2 Scheme of VCS,

Source: <https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control>

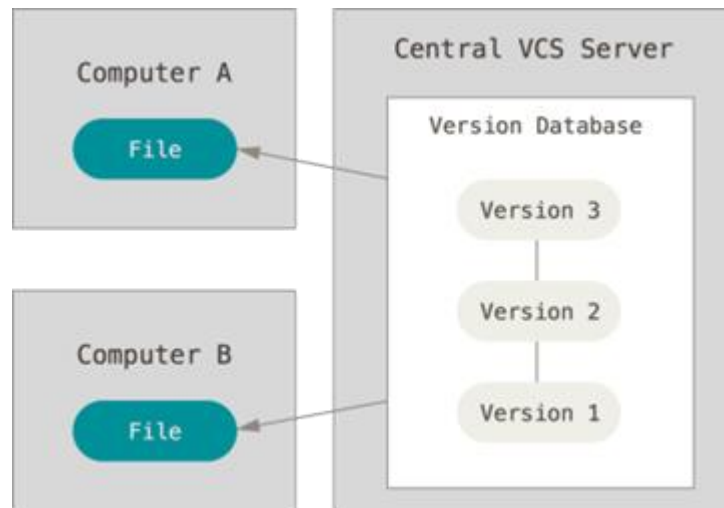


Figure 2.3 Scheme of CVCS,

Source: <https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control>

The most common issue with CVSC is the point of failure. It means, that everyone working on the project is dependent on the server holding the files. To overcome this problem, Distributed Version Control Systems (DVCS) were introduced. In a DVCS, clients fully mirror the repository, including its whole history, as opposed to only checking out the most recent snapshot of the contents. Therefore, any client repository may be transferred back up to the server to recover it if any server goes down and these systems were cooperating through that server. Every copy serves as a complete backup of all the data [5]. Figure 2.4 shows the scheme of such a system.

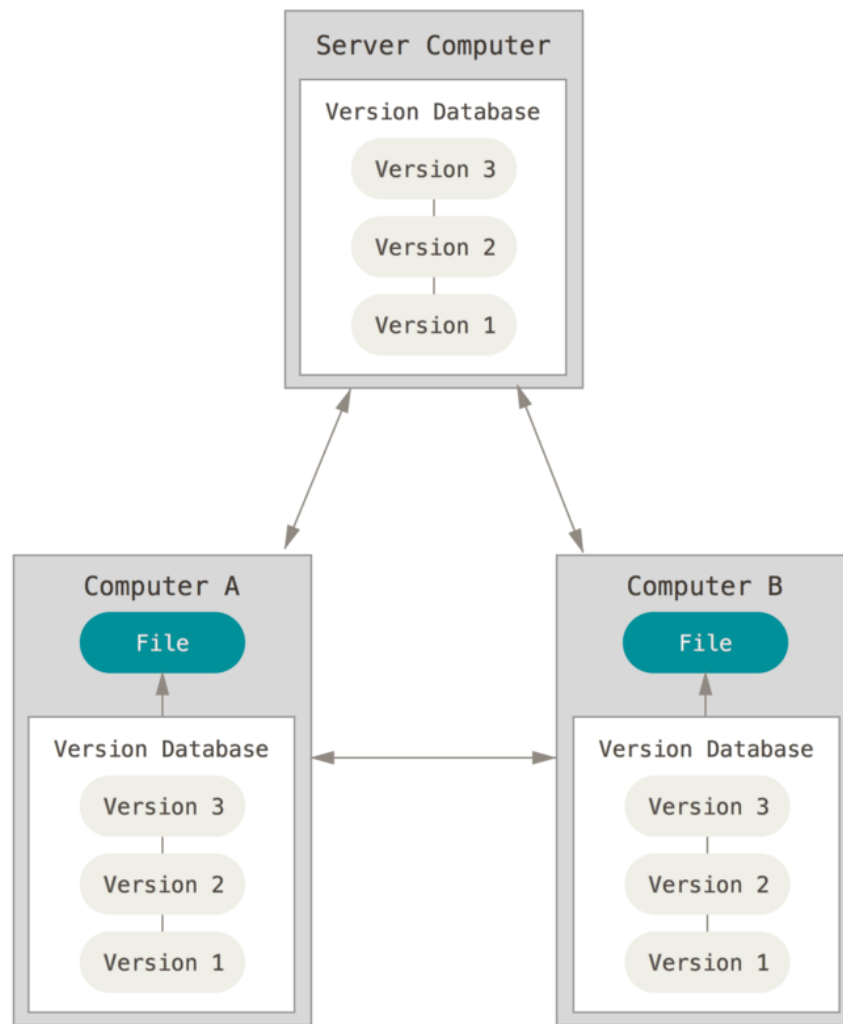


Figure 2.4 Scheme of DVCS,

Source: <https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control>

By far the most popular VCS system is Git, which initially was created to help the development of the Linux kernel. The place where files are stored under Git is called the repository. What makes Git different from other VCSs is how it sees the files. Instead of tracking changes in each file over time, Git views its data more as an ongoing series of snapshots. It records a reference to the snapshot it takes of the state of all your files at that precise instant. Git only stores links to previously saved similar files when files haven't changed to be as efficient as possible [5].

Files in Git can be in three stages:

- Modified – when the change was made, but it wasn't committed yet;
- Staged – when the file was marked as modified in the actual version for the next snapshot;
- Committed – when the data is stored in the local repository.

Besides that, git has numerous useful features, but for simplicity two mainly used in this project, will be described. These are branching and forking. When a user creates a new branch, Git is creating a pointer to the current commit. This operation allows the introduction of new changes without inflicting the main branch of the project. After work is done on the branch, it can be discarded or merged into the mainstream of the project after reviewing it and making sure it won't break anything. Forking, on the other hand, is creating a new repository from the existing one. It creates an identical repository, which is particularly useful if there's a need of modifying or experimenting freely with a publicly available repository, but a user doesn't have the rights to publish without any limitations to it.

As mentioned above Git can store files on the server and then locally on each developer's machine. There are many different implementations of git and the most widely used server is GitHub. It allows creation of public repositories without limitations, or private ones, which are available for free only to some extent. Furthermore, GitHub provides the possibility of convenient creating issues. GitHub Issues is a planning tool that makes it easier to focus on important tasks, creating pull requests, which is an event where a contributor asks an administrator of the repository to review the changes he made and merges it into the project and many more which are outside of the scope of this thesis. Communication with the server can happen with the use of different protocols and authorization or authentication methods, making communication safe and convenient for users or other systems that need to interact with the repository.

The main tool to communicate with the Git repository is the command line tool (CLI). Nevertheless, upon gaining popularity many different integrated development environments (IDE) integrated Git into their graphical user interface (GUI), or completely separated solutions were created to handle repositories. They make working with Git easier, but behind the scenes, they are invoking the same commands the developer would execute while working with the command line.

Example workflow of working with Git and GitHub, with simplified commands is presented below:

1. *git clone* – clones repository from the server to the local machine when a developer starts to work on the project for the first time;
2. *git pull* – it pulls the actual version of the repository from the server, ensuring that the most recent version is present on the local machine;
3. *git branch* – creates a new branch that the developer will work on, and then switches to it;
4. Developer makes changes to the files;
5. *git add* – adds selected and modified files into the staged stage;

6. *git commit* – adds staged files to the committed ones;
7. *git push* – publishes the branch to the server, simultaneously creating a new pull request;
8. Maintainer of the repository decides if the newly created branch should be merged into the project.

With the knowledge about storing and versioning files inside the repository, there must be a way of a convenient, reliable, fast, and easy way to deliver the developed product to the customer. One of the methods that fit this description is called Continuous Integration and Continuous Delivery, which will be the next topic of this thesis.

2.3. Continuous Integration Continuous Delivery

A successful project needs to adapt to changes to meet business needs. Apart from that, after the application's initial release, it must be constantly enhanced or extended with new features. Without automation, deployment would take a long time, foremost it would require a lot of manual work. Each time update is made on a minor piece of code, there would be a need for redeploying it. After adding modifications to the project, CI/CD allows to automate the integration and deployment process with a single button click or even without it, depending on project configuration [6].

In CI/CD, the first part stands for Continuous Integration, while the second stands for Continuous delivery or continuous deployments. CI/CD is a means of delivering apps to clients more often by incorporating automation into the app development process. CI/CD, in particular, adds continuous automation and monitoring across the app lifecycle, from integration and testing through delivery and deployment. These approaches are referred to be a "CI/CD pipeline" when they are combined, and they are supported by development and operations teams working together in an agile style using a DevOps mindset [7].

The first part of the discussed acronym is continuous integration, which considers mainly development teams. The objective of contemporary application development is to have numerous developers working on various parts of the same app at the same time. Daily, however, the effort of merging all branching source code can be arduous, manual, and time-consuming. The reason behind this is that, when a developer alone makes a change to an application, there's a danger it'll conflict with other modifications being made at the same time by other developers. Developers use CI to integrate their code changes to a common branch more often, sometimes even daily. After modifications to an application are merged, the changes are validated by automatically building the app and executing several automated

tests, which results in checking the app's modules to ensure that the program works as expected.

On the other hand, there is a CD part that can have two separate meanings. The first of them is Continuous Delivery. Continuous delivery automates the release of that verified code once continuous integration has done its job. The purpose of continuous delivery is to have a repository with code ready to be deployed to a production environment at any time. The second meaning is continuous deployment, which is the last stage of the CI/CD pipeline. Continuous deployment is an extension of continuous delivery. It serves the purpose of automating the release of an app to production. Continuous deployment, in effect, implies that a developer's contribution to a cloud application might be live minutes after it was written if the tests pass. The representation of the CI/CD pipeline is shown in Figure 2.5.



Figure 2.5 Stages of CI/CD,

Source: <https://www.redhat.com/en/topics/devops/what-is-ci-cd>

The main reason why CI/CD was created is to overcome the issue known as “integration hell”. It is one of the most difficult problems in software development environments. The integration process, especially in bigger projects, is always a challenging task, and it often takes hours, if not days, to re-fix the code before it can eventually merge. One of the most difficult issues is the dependence connection between the created code and the rest of the system's code. Apart from that, each process that can be automated will result in saving time in the bigger picture.

Having in mind two methodologies for developing and delivering the product to the users, the next question that arises is: “In what form application should be delivered?”. The answer to this question is containerization, which leads to the next chapter.

2.4. Containerization

Organizations across the world are progressively adopting containers. As of January 2021, over 3.5 billion apps are presently running in Docker containers, and 48 percent of enterprises use Kubernetes to manage containers at scale, according to Research and Markets

[8]. These two statistics alone show the importance and ongoing trend in the industry. It also proves that containers are playing a big role in the modern information technology (IT) world.

Before containers became so popular, the main technology behind hosting web applications was virtualization. A virtual machine (VM) is an emulation of a physical computer. VMs allow many operating systems to operate on a single server, allowing enterprise applications to use resources more efficiently. A software layer, known as a hypervisor, manages virtual machines, isolating them from one another and allocating hardware resources to each of them like Central Processing Units (CPUs), memory, storage, and networking [9].

Containerization is a type of system virtualization in which just the services and packages that are necessary are deployed on a minimal operating system (OS). Because these resources are all self-contained, working with and maintaining their dependencies is simplified. Container-based apps can be deployed easily and consistently regardless of the target environment since programs can be separated from the environment in which they execute [10]. Like VM, multiple containers can run on the same machine and share the operating system's kernel, each executing as a separate process in user space [11]. A very important attribute of containers, which needs to be highlighted, is that they run in an isolated environment. In practice, it means that, unless specified otherwise, two separated, started containers don't know about each other's existence. The difference between containerization and virtualization is shown in Figure 2.6.

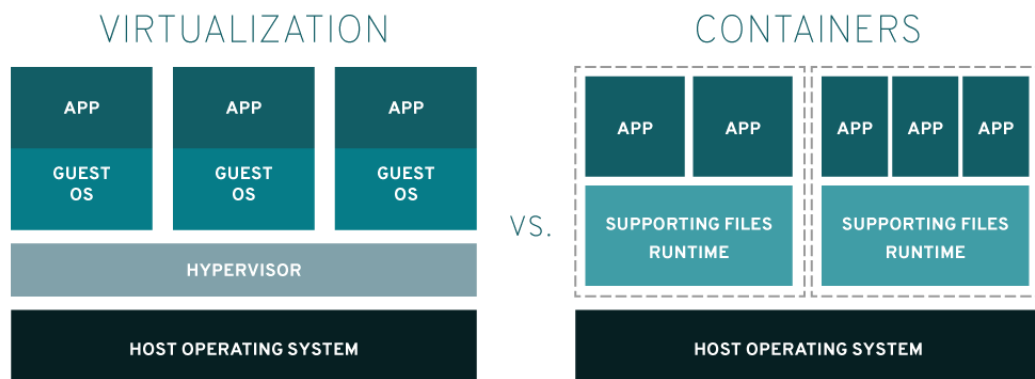


Figure 2.6 *Difference between virtualization and containerization,*
Source: <https://www.redhat.com/en/topics/containers/containers-vs-vms>

As these two technologies share similarities, performance-wise containerization wins over virtualization. While virtual machines suffer from a small overhead as the machine instructions are translated from guest to host OS, containers provide near-native performance as compared to the underlying host OS [9],[10]. Also, containers can be booted up in a few seconds, as compared to VMs, which might take even a few minutes. As per storage occupied

VM takes much more of it as the whole OS kernel and associated programs must be installed and run compared to containers which don't require that amount of space due to the shared base operating system [9]. Although containers have better performance results, they suit different purposes. A very common practice in the public cloud is to run containerized applications on a virtual machine.

Containers, like virtual machines, also require a layer responsible for managing them. This technology is called a container engine. The most widely used container engine is undoubtedly Docker, but since the words docker and container are used almost synonymously in many situations, thus, it is important to distinguish separate parts of a container engine, which will be done with Docker as an example. The knowledge of how different parts of the container engine work with each other will be important while discussing the container orchestrator in further chapters.

On top of the scheme shown in Figure 2.7, there is a client, in this case, docker (or docker-cli). This is the place where a user performs actions on containers, like creating, starting, or stopping the container. Then, docker communicates with high-level container runtime which is containerd. It's a daemon process responsible for managing containers, storage, or networking and also pulls or pushes images. After that, containerd communicates with low-level container runtime that creates and runs containers. This component is called runC. When containers started to revolutionize the world, Docker was the only container engine out there, but it's not the case today. For this reason, there was a need for the specification of container images and running containers. This is how Open Container Initiative (OCI) was created. Besides container specification, it also provides an implementation of the runC.

Like the project's files, developed containers are also stored and versioned in a place called the container registry. This is a repository, or group of repositories, for container image storage and access. Container registries can assist with the development of container-based applications, frequently as part of DevOps procedures. Apart from that, they play an important role in container orchestration, which will be the topic of the next chapter, by allowing them to pull the images.

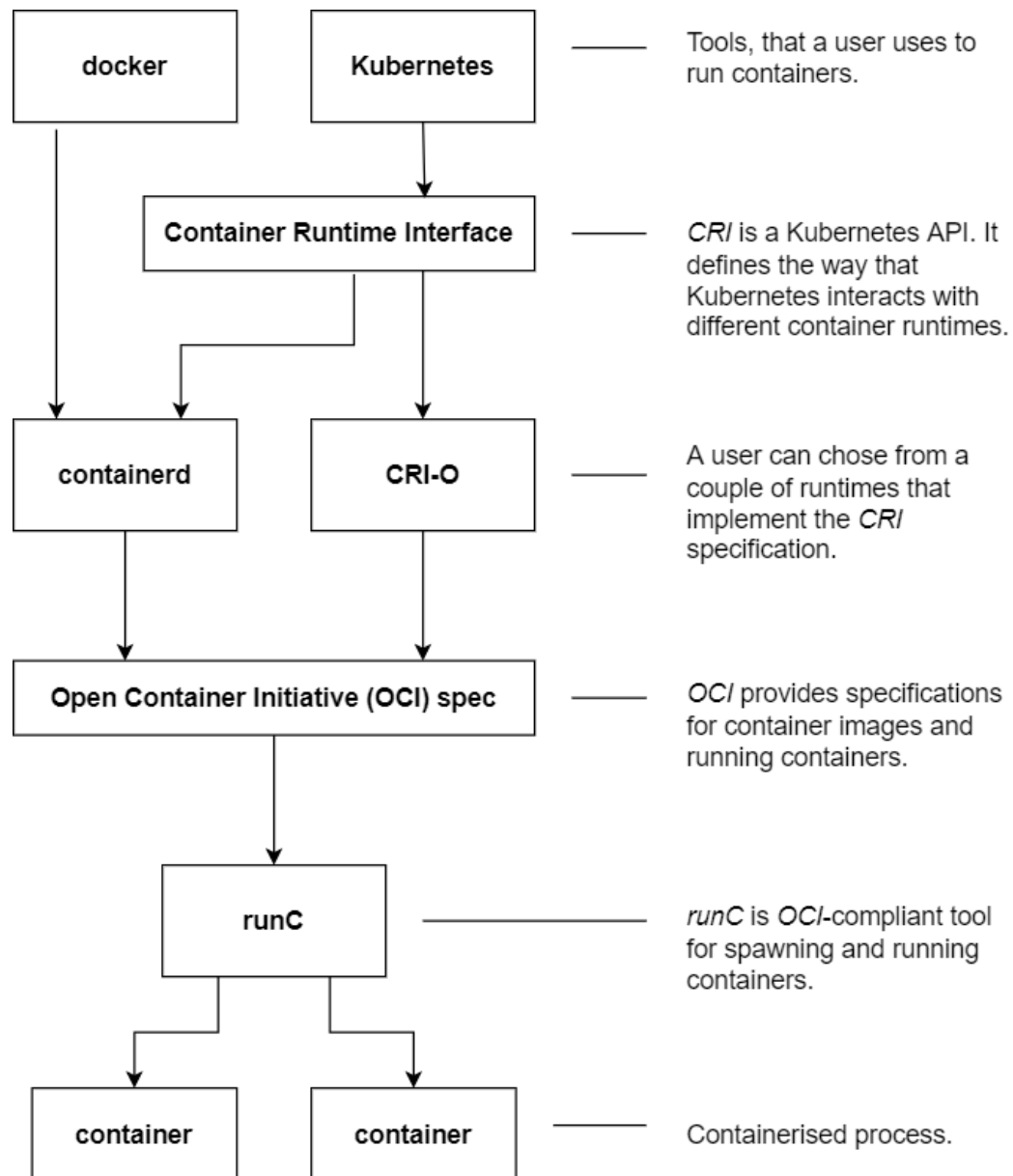


Figure 2.7 The relationship between Docker, CRI-O, containerd, and runC,
 Source: <https://www.tutorialworks.com/difference-docker-containerd-runc-crio-oci/>

2.5. Container Orchestration

When the project grows and becomes bigger, a single environment can consist of tens, hundreds, or even thousands of containers and virtual machines. When this happens, the task of managing such infrastructure becomes a challenging task. Harder becomes creating underlying communication between services, their security, and load balancing which is the way of efficiently distributing incoming network traffic across a group of services. To solve these problems container orchestration tools were created.

Container orchestration automates container deployment, management, scaling, and networking, as well as resource scheduling, microservice coordination, communication, and resource booking. Container orchestration works in any context where containers are employed. It assists in deploying the same program across several environments without having to rewrite it. Containers provide a suitable application deployment unit and self-contained execution environment for microservice-based apps. They let many elements of an app run independently in microservices on the same hardware, giving developers far more control over individual components and life cycles. Using orchestration to manage the lifetime of containers simplifies the work of DevOps teams that integrate it into CI/CD workflows [12],[13].

Container orchestration technologies offer a framework for controlling containers and microservices architecture in a scalable fashion. Container lifecycle management can be accomplished with a variety of container orchestration solutions. Kubernetes, Docker Swarm, Apache Mesos, and Cattle are examples of them. From the mentioned ones, undoubtedly, the most popular solution on the market is Kubernetes (k8s). Its name comes from the Greek word which means the word “pilot” or “helmsman”. Initially, it was released by Google LLC in 2014, and in 2016 it was donated to the Cloud Native Computing Foundation (CNCF) as the first tool to do so. Kubernetes is an open-source project distributed under Apache License 2.0. Nevertheless, in recent years solution became so popular that many organizations on the market adopted them to better suit their and their user needs, mainly in the cloud. Examples of such distributions are Elastic Kubernetes Engine (EKS), developed by Amazon, Azure Kubernetes Service (AKS), developed by Microsoft, or Google Kubernetes Engine (GKE), maintained by Google.

As this thesis focuses on Kubernetes as a container orchestrator, its architecture will be described. Instead of directly managing containers, Kubernetes encapsulates them into pods. These are the smallest and the most basic deployable objects in k8s. Kubernetes deployment is called a cluster and consists of two main components, which are the control plane (also called the main plane, a master node) and node (also called workers or worker nodes). Both of them consist of different components that have different purposes [14],[15]. The scheme of the Kubernetes cluster with all its elements is presented in Figure 2.8.

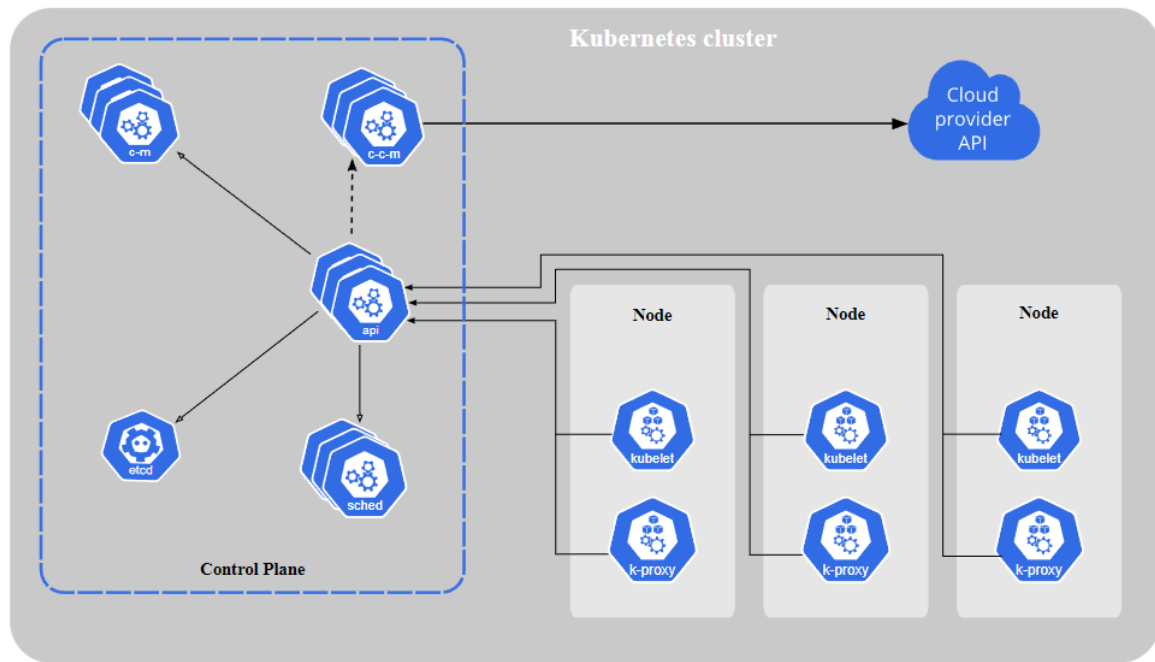


Figure 2.8 *Kubernetes Components*,
 Source: <https://kubernetes.io/docs/concepts/overview/components/>

Control plane components are:

- *kube-apiserver* – exposes Kubernetes application programming interface (API) and it's the front end for the Kubernetes master nodes [15];
- *etcd* – it's an open-source, consistent, distributed key-value store that gives a safe way to store data accessed by a distributed system. K8s uses it as a backing store for all cluster data [15],[16];
- *kube-scheduler* – its task is to watch for new Pods, which consist of containers and handle workload inside the Kubernetes, with no assigned worker node, and then select the node where the pod will run [15];
- *kube-controller* – it runs controller processes, where each controller process is separated from another one. Example controllers are node controller, job controller, or endpoints controller [15];
- *cloud-controller-manager* – it embeds cloud-specific logic, meaning it allows linkage of the cluster into the cloud provider's API. Besides that, it separates components that are interacting with the cloud from ones that interact only with the cluster [15].

Node components are:

- *kubelet* – it's an agent that must run on every worker node in a cluster. Its task is to ensure that containers are running correctly in the pod;

- *kube-proxy* – is another component of worker nodes that needs to be present on each of them. It's a network proxy that maintains network rules on nodes, thus, allowing communication inside and outside of the cluster;
- *Container runtime* – as noted earlier, it's a component responsible for running containers.

As mentioned before, container runtime for Docker is containerd. At the beginning of the Kubernetes project, k8s allowed compatibility with only one container runtime, which at the time was Docker. It was possible through *docker-shim* which was a bridge between Docker and Kubernetes. Then to allow other container runtimes the Container Runtime Interface (CRI) was created. It defines the way how Kubernetes interacts with different container runtimes. A very notable moment in Kubernetes development happened on 02.12.2020 when its developers announced the deprecation of Docker as its container runtime [17]. A scheme that illustrates this change is shown in Figure 2.9. It then finally happened with the release of version 1.24 and caused a big shift from the Docker runtime to containerd. According to Datadog [18] only between January 2021 and July 2021 usage of containerd rose from almost 0% to 6% across the preferred Kubernetes container runtime and it is expected to grow even more rapidly in the following years.

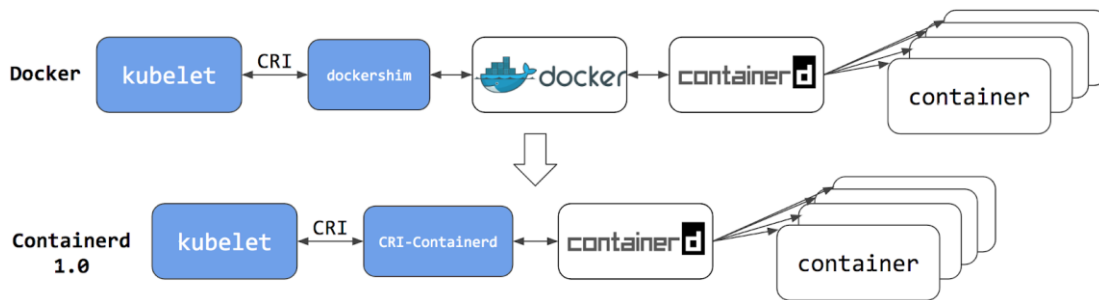


Figure 2.9 Change after deprecation of dockershim,
 Source: <https://kubernetes.io/docs/tasks/administer-cluster/migrating-from-dockershim/check-if-dockershim-removal-affects-you/>

To show the importance and scale of the Kubernetes, another statistic will be brought. According to Datadog in November 2020, more than half of organizations were using some kind of k8s distribution. The same report tells that nearly 90% of containers are orchestrated [19]. Both statistics also have a growing tendency as presented in Figure 2.10. All these numbers reveal how industry and market shifted from the traditional way of running applications to a container-based with an orchestrator on top of it.

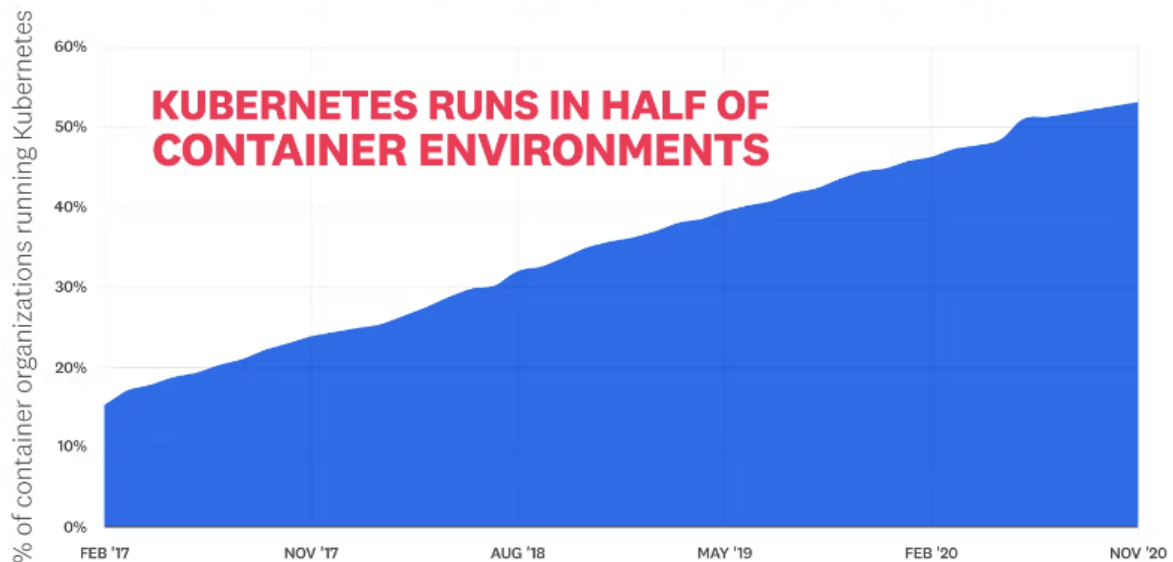


Figure 2.10 *Kubernetes share among container organizations*,
Source: <https://www.datadoghq.com/container-report-2020/>

Despite all its benefits and popularity, container-based architecture still requires physical infrastructure to run on. The perfect solution for this is the public cloud. Its benefits, along with the different models are described in the next section of the thesis.

2.6. Public Cloud

Cloud computing refers to the distribution of computing services over the Internet, including virtual machines, storage, networking, analytics, databases (DB), software, and intelligence, to provide faster innovation, more flexible resources, and economies of scale. Apart from traditional payment models, cloud providers offer Pay-as-you-go pricing, which in practice means that users pay only for the resources they are using.

Cloud computing offers four main models of “as a service” and each one covers a different degree of management for the user, [20] which are:

- Infrastructure as a service (IaaS) – a model where a provider gives access to management of the network, servers, virtualization, and storage that is required, but the user is responsible for the operating system and any data applications, middleware, and runtimes. Because the supplier updates the data center, there is no need for the customers to do so. IaaS allows freedom to acquire only needed components and grow them according to the requirements. The main disadvantages of IaaS include the likelihood of the provider’s security issues, resulting from the fact that the provider must share resources with several

customers. By selecting a reputable and trustworthy source, these drawbacks can be avoided [20].

- Platform as a service (PaaS) – a full-featured cloud development and deployment environment that has the tools needed to produce everything from basic cloud-based apps to complex corporate applications. Along with infrastructure (storage, servers, and networking), PaaS also comprises development tools, middleware, business intelligence (BI) services, DB management systems, etc. The full lifetime of a web application, including development, testing, deployment, management, and upgrading, is supported by PaaS. Normally while the user manages the apps and developed services, the cloud provider manages everything else [21].
- Software as a Service (SaaS) – enables to purchase of complete software solutions from a cloud service provider. Users connect to an app through the Internet, typically using a web browser. The data center of the service provider holds all of the supporting infrastructure, middleware, app software, and app data. With the right service agreement, the service provider will also guarantee the app's availability, security, and privacy. The service provider manages the hardware and software [22].
- Serverless computing – by removing the requirement for infrastructure management, serverless computing enables developers to create applications more quickly. In serverless applications, the infrastructure needed to run the code is automatically provisioned, scaled, and managed by the cloud service provider. It's vital to remember that servers are still running the code despite the name. The name "serverless" refers to the notion that infrastructure provisioning and management duties are hidden from developers. With this strategy, developers may concentrate more on business logic and add more value to the organization's core [23].

Differences between models are shown in Figure 2.11.

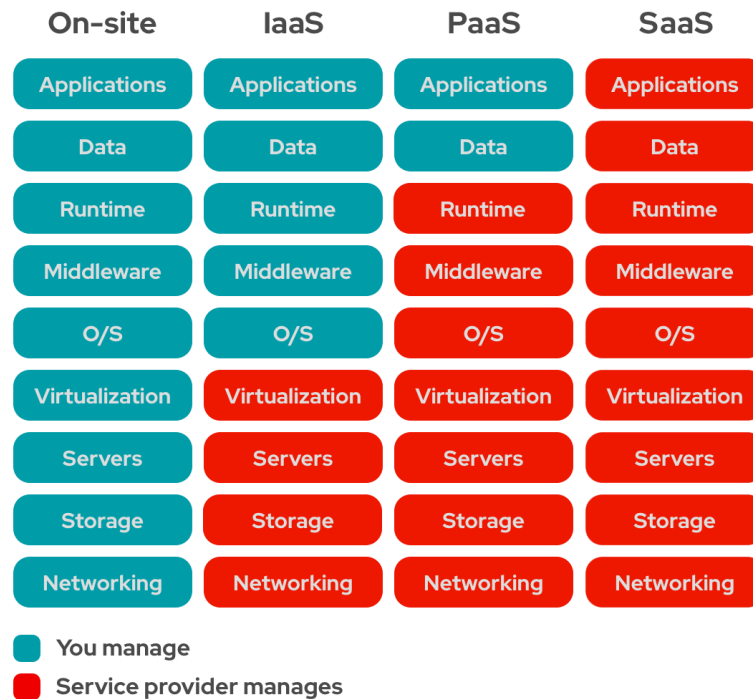


Figure 2.11 *Difference between deployment models*,
 Source: <https://www.redhat.com/en/topics/cloud-computing/iaas-vs-paas-vs-saas>

Apart from the different types of service models, there are also different categories of deployment models. These are:

- **Public cloud** – this term refers to computing services made available to anybody who wishes to use or buy them via the public Internet. Free or on-demand sales options are available, allowing users to pay only for the CPU cycles, bandwidth, storage used, etc.
- **Private cloud** – the type of cloud where computing services are delivered only to a specified group of customers via a private network or the public Internet. Also known as the internal or corporate cloud, offers businesses many of the advantages of a public cloud, such as self-service, scalability, and elasticity, with the additional control and customization available from dedicated resources over an on-premises computing infrastructure. Due to this, security and privacy are improved, by involving the organization's firewalls and internal hosting [24].
- **Hybrid cloud** – it is a combination of public and private clouds, which are connected, thus, enabling the transfer of data and applications between them. This model provides more flexibility and aid in the optimization of infrastructure, by enabling data and apps to flow between both mentioned earlier models.

Cloud computing comes with many different benefits for users. The most important one is probably the costs. With the cloud, there is no need for buying hardware or software and creating on-premises data centers. All the elements of the physical infrastructure are managed by a cloud provider. Another advantage is that the cloud allows to spin-up of most resources within minutes. Undoubtedly, the ease with which provisioning can be done plays an important role, since with just a few mouse clicks there is a possibility of provisioning vast amounts of servers with, automatically created, private network or Internet Protocol (IP) addresses. Performance, reliability, and security cannot be forgotten in the context of the benefits the cloud gives. Cloud providers are taking all necessary measures to provide these features. Their data centers are run on a worldwide network of the secure data center, which receives regular updates to ensure the best computing hardware. Also, cloud providers make data backup, and offer disaster recovery mechanisms and a wide set of policies or controls that enhances security.

In the context of Public Cloud Providers, three main companies come to mind. These are Amazon with Amazon Web Services (AWS), Microsoft with Azure, and Google with Google Cloud Platform (GCP). According to the Synergy Research Group [25], mentioned providers in the fourth quarter of 2021 were responsible for 54% revenue of global market share, which for the first time exceeded 50 billion US dollars and in the whole of 2021 was equal to 178 billion US dollars. Figure 2.12 presents the distribution of the global market share between the biggest cloud providers and Figure 2.13 presents the trend in the market share. These statistics might differ from research to research, but these three names always come on top. What is important to note is that these statistics include IaaS, PaaS, and hosted private cloud services. Nevertheless, if only enterprise SaaS will be taken into account different names will be listed. Enterprise SaaS means applications suited for big organizations. Vendors of such are for example Microsoft with Office 365, Salesforce with their Customer Relation Management (CRM) platform called Sales Cloud, or SAP with Enterprise Resource Planning (ERP) system named S4/HANA. These will not be analyzed in further work; thus, this thesis will focus on AWS, Azure, and GCP.

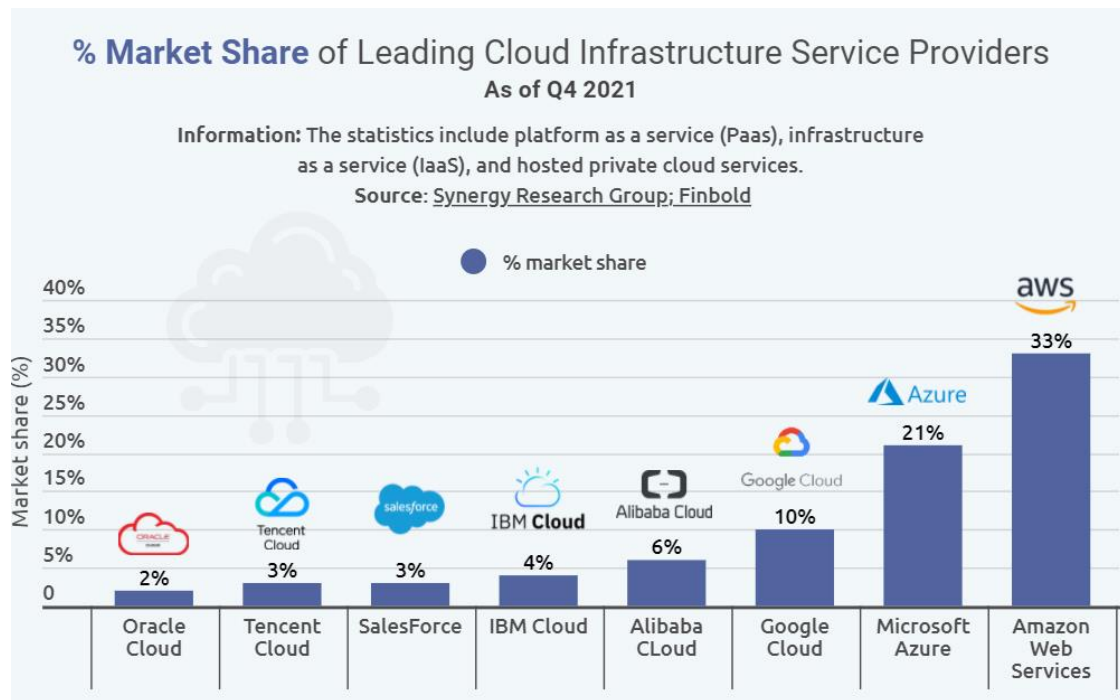


Figure 2.12 Market Share of leading cloud infrastructure providers as of Q4 2021,
Source: <https://www.financialmirror.com/2022/02/11/amazon-aws-clinches-33-cloud-market-share/>

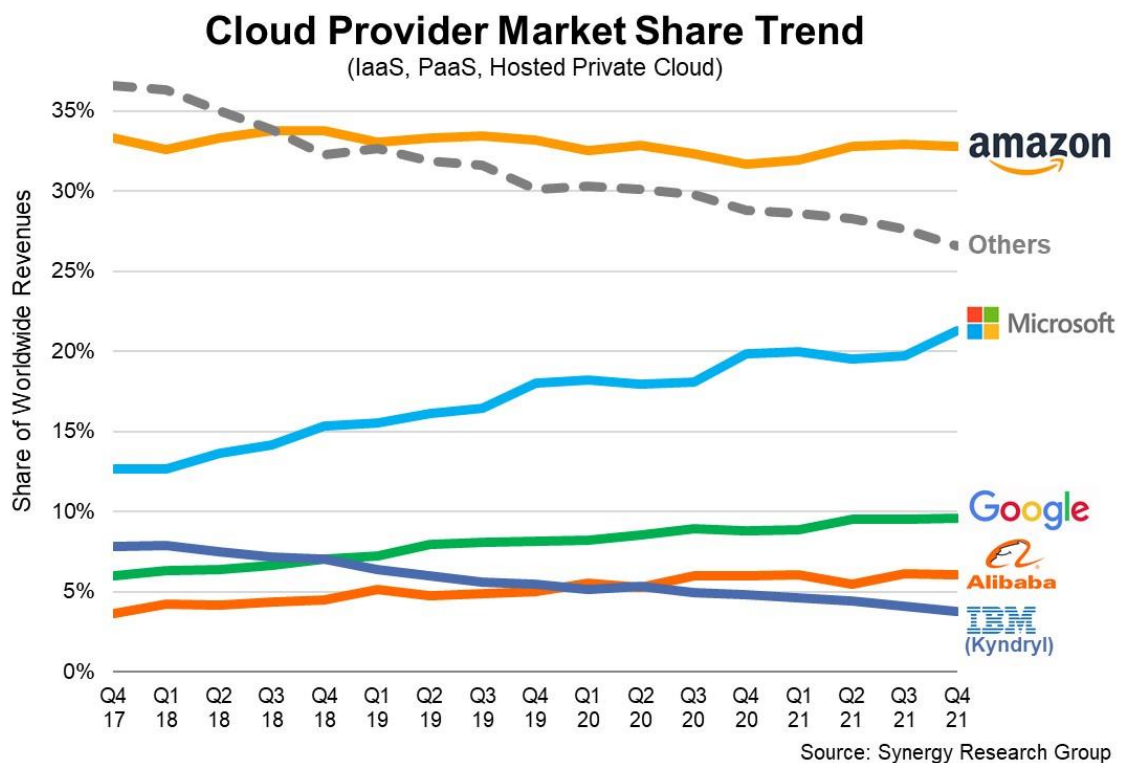


Figure 2.13 Cloud Provider Market Share Trend,
Source: <https://www.srgresearch.com/articles/as-quarterly-cloud-spending-jumps-to-over-50b-microsoft-looms-larger-in-amazons-rear-mirror>

3. Platform and Technologies

DevOps methodology can be applied to almost every software development project. This means that the number of available tools, platforms, or technologies can be overwhelming. For this reason, the task of choosing the most appropriate technology stack for the given project is often very challenging. The fact, that solutions are also divided into open-source and paid ones, is not helpful at all, since apart from the tool's functionality, the designing engineers must also focus on the price limitations.

This chapter will focus on the journey of selecting the most appropriate tools and platforms for the project. It will start from the description of the sample application and its components. It will be followed by describing the biggest cloud providers on the market along with their products. Then, build tool accompanied by an automation server will be chosen. The chapter will finish by selecting each needed component to automate the deployment process of the application, based on the research, price, and performed experiments.

3.1. Application

To present and explain benefits in a demonstrative way, that come from introducing DevOps and automation into SDLC, the project must meet certain requirements. In this paper assumptions regarding the used project are as follows:

- It is under version control, to allow changes to the code;
- It must be deployable to the servers;
- It must contain tests, e.g., unit or integration tests.

Many different programming languages can fulfill these needs. Nevertheless, a Java-based web application will be used, as it's still a very popular language in the development of microservices. The application is built on top of the Spring Boot Framework with Tomcat as an application server. It's stored under GitHub Version Control and uses JUnit as a framework for writing and performing unit tests.

Before explaining how the project is built it is important to know the purpose of the key technologies behind its implementation:

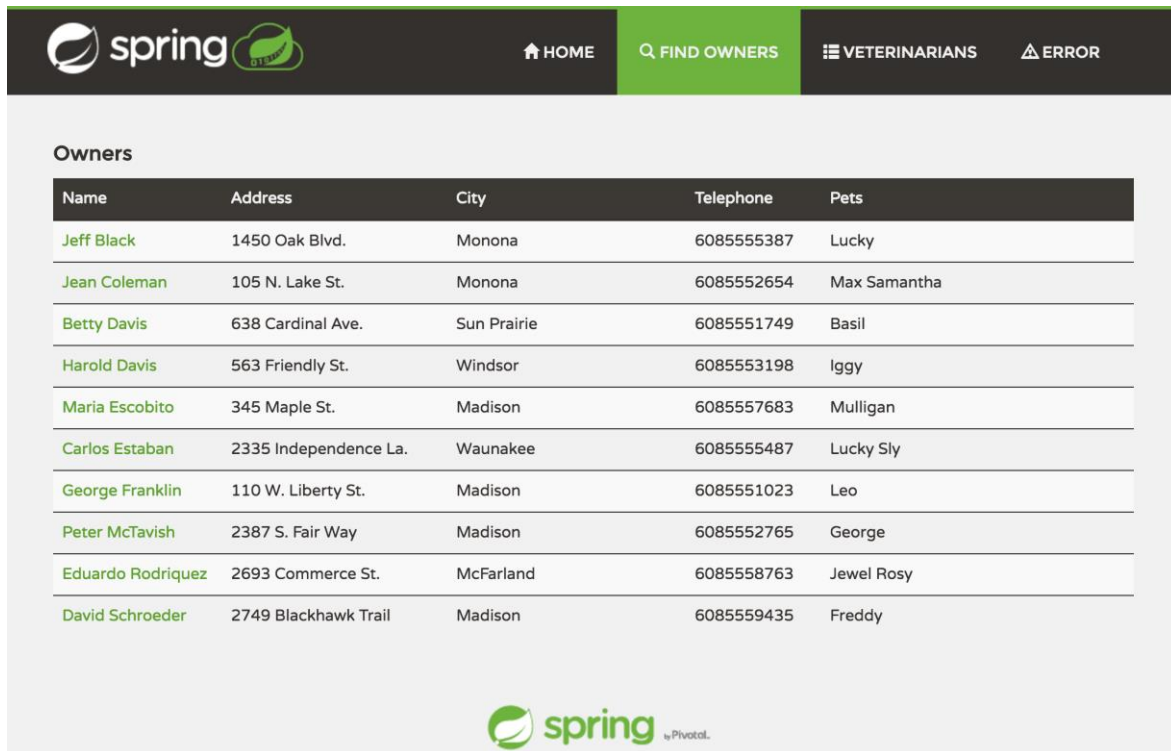
- Java is an object-oriented, general-purpose programming language with a class-based architecture that is intended to have fewer implementation requirements. It

is an application development platform for computers. Java is consequently quick, safe, and trustworthy. Java is also cross-platform and the main intention of its creators was to be able to “write once, run anywhere”.

- Java Spring Framework often referred to simply as Spring Framework, is an open-source, enterprise-level framework for building independent, high-quality Java Virtual Machine applications (JVM).
- Java Spring Boot Framework, referred to as Spring Boot, is also a framework for Java programming language. It is built on top of Spring Framework and allows to develop microservices easier and faster through its core capabilities, which are: autoconfiguration, taking a position on configuration, and ability to create standalone applications.
- Apache Tomcat is an open-source Java servlet container that provides the Java Servlet, JavaServer Pages (JSP), and WebSockets Application Programming Interfaces (APIs), among other important Java industry specifications. To put it simply, Tomcat serves as an application server that is specifically designed to run applications.
- JUnit is a framework for unit testing in the Java programming language and is essential to test-driven development. Testing is checking an application's functionality to make sure it operates in accordance with, requirements where unit testing is used to test a single entity (class or method). Apart from that unit testing can be performed in two different ways – manual or automated testing.

As this thesis focuses on the automation of parts of the software development life cycle, instead of going into programming nuances, already developed, an example project was used [26]. The application is called “The Spring PetClinic”, with many different people contributing to it and it’s released under Apache License 2.0 [27]. The project, besides application source files, contains a lot of useful resources, like build tools wrappers, configuration files for different build tools, etc.

The application is a sample web app for pet clinic, where the clinic administrator can register patients with their owners. For storing its data, the h2 in-memory database is used. The usage of an in-memory database means that data is stored in memory instead of storing data on the disk. It is convenient because with this method the application can run itself without any additional components. The biggest drawback of this solution is that any changes to the database made during the working of the program are not persistent, meaning that after restarting the database will be reverted to the initial state. Nevertheless, as this application is only for testing purposes, this simplification is an advantage and an easy way to overcome this obstacle is to populate the database with testing data during startup. An example screenshot from the application is presented in Figure 3.1.



The screenshot shows the Spring PetClinic application interface. At the top is a navigation bar with the Spring logo, a search bar, and links for HOME, FIND OWNERS, VETERINARIANS, and ERROR. Below the navigation bar is a section titled "Owners" containing a table with owner information.

Name	Address	City	Telephone	Pets
Jeff Black	1450 Oak Blvd.	Monona	6085555387	Lucky
Jean Coleman	105 N. Lake St.	Monona	6085552654	Max Samantha
Betty Davis	638 Cardinal Ave.	Sun Prairie	6085551749	Basil
Harold Davis	563 Friendly St.	Windsor	6085553198	Iggy
Maria Escobito	345 Maple St.	Madison	6085557683	Mulligan
Carlos Estaban	2335 Independence La.	Waunakee	6085555487	Lucky Sly
George Franklin	110 W. Liberty St.	Madison	6085551023	Leo
Peter McTavish	2387 S. Fair Way	Madison	6085552765	George
Eduardo Rodriguez	2693 Commerce St.	McFarland	6085558763	Jewel Rosy
David Schroeder	2749 Blackhawk Trail	Madison	6085559435	Freddy

At the bottom of the page is the Spring logo and the text "by Pivotal".

Figure 3.1 Example view from Spring “PetClinic” application,
Source: <https://github.com/spring-projects/spring-petclinic>

It is worth noting that during the deployment of the application, another simplification, besides the in-memory database, will be used. Namely, the lack of a web server, among which the most popular is the Apache HTTP Server. Nowadays, to deliver content to the client, a commonly used method is a three-tier architecture, which is presented in Figure 3.2.

In the three-tier architecture, the client or presentation layer is considered to be the first. On this client tier, the Apache HTTP Server is located. It provides a final answer back to the client and is the first server-side resource a client interacts with while making a request. The Apache web server will provide a basic, static file to the client in response to a request for one, such as an image or HyperText Markup Language (HTML) document. When a request necessitates some logic, the Apache web server forwards it to the Tomcat server. The Tomcat server is therefore seen as being a component of the middle tier of a three-tier architecture. The last tier is the data tier, also called the database tier, back-end, or data access tier. It is the place where an application’s information is processed, managed, and stored [28].

Three tier architecture

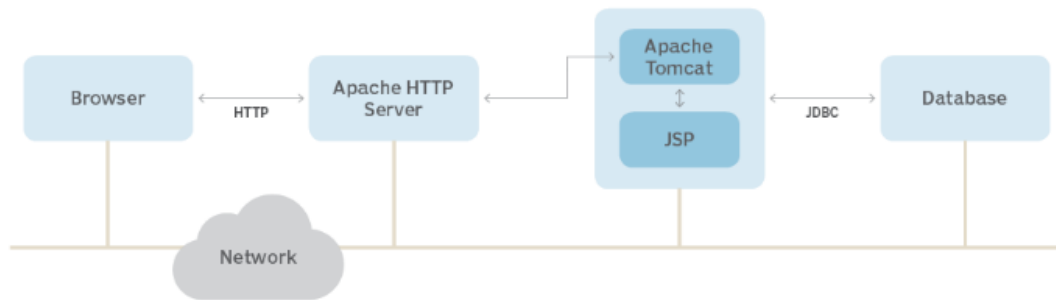


Figure 3.2 *Three-tier architecture,*

Source: <https://www.theserverside.com/video/Tomcat-vs-Apache-HTTP-Server-Whats-the-difference>

Having an application that can be deployed to the servers and understanding how it works, the next step is to choose the correct infrastructure it can be deployed to. As said in chapter 2.6 public cloud gives numerous benefits, which leads to the task of choosing the most appropriate cloud provider for the project. The task will be carried out in the next chapter.

3.2. Cloud Providers

While talking about deployment automation, environments are often split into two types – ones that are responsible for automation processes and target environments. This practice will be followed in this work. As shown before, the market is dominated by three main cloud providers which are AWS, Azure, and GCP, so in the beginning, the choice will be narrowed to these three.

Looking at the most basic services like virtual machines, IP addresses, virtual networks, or disks, their offer is quite similar. A much larger difference is noticeable during the selection of the more specific services like Artificial Intelligence (AI) processing, data lakes, or Internet of Things (IoT) infrastructure, to mention a few. None of the more specialized solutions will be needed though. Apart from the offered services they also differ in pricing, or location of data centers, which is important due to latency, if a project is not implementing a content delivery network (CDN).

Before choosing the best provider to suit the project's needs, an important task must be carried out, which is realizing what is needed and what are the limitations. Not all projects are financed by big organizations with large investing possibilities. Thus, it is assumed that the project should consume as little money as possible while maintaining usability since it will mimic the development environment. Also, all end users working on it will be located in Poland. Another assumption is that hypothetical engineers working for the project are not very experienced with the cloud, so ease of working with each of the cloud providers will be considered. That being said, Amazon Web Services, Azure, and Google Cloud Platform will be compared based on the price, latency of the requests, and ease of implementation.

Firstly, platforms responsible for storing and versioning the project and created containers will be chosen. Each of the cloud providers offers their alternative to GitHub, but due to its popularity, great integration with other tools, and the fact that the original "Petclinic" project is developed with the use of GitHub, this will be the VSC in this work.

Regarding the container registry, discussed cloud providers also offer their solutions available within their cloud. Nevertheless, these are usually paid solutions, meant for commercial use. As this project doesn't require private repositories for the containers, the most popular, free alternative, called Docker Hub, will be chosen. Besides the fact that the solution is free, it is also a product from the same company that develops Docker. This results in great integration between Docker Engine and Docker Hub.

Another important step that must be taken is to analyze what components are needed for each environment. Both will need a server to host applications. To fulfill this, the main component for environments will be virtual machines. Apart from that, each virtual machine needs a disk, to install the OS and store files and a public IP address to provide access to them. To allow communication between VMs, a virtual network must be created. For ensuring a basic level of security service called Network Security Group (NSG) will be used. Apart from the mentioned resources target environments will require a Kubernetes cluster. There are two available options: creating a cluster manually or using a cluster managed by the cloud provider. According to Datadog [19], almost 90% of organizations use cloud-managed Kubernetes clusters. This trend will be followed in the further read; thus, only managed clusters will be reviewed.

The next decision that needs to be made is what types of mentioned resources are needed. Each of the cloud providers offers different types of virtual machines or disks. VMs can have different sizes regarding computing power or memory. Virtual machines are also optimized for different purposes like General Purpose, Compute optimized, Memory optimized, etc. [29]. Besides that, not all operating systems are free. For example, most Linux distributions are free in opposite to Windows or macOS for which users must pay. For that reason, on each virtual machine, Ubuntu 20.04 will be installed whenever it's possible.

Disk's performance is measured in different metrics. Except its capacity, the main ones are input/output operations per second (IOPS), which is an evaluation of the maximum number of reads and writes to storage a location, and throughput, being a measurement of how fast storage can read or write data. Throughput is usually written in megabytes (MB) per second. It is important to keep in mind that these metrics depend on the disk size, number of virtual CPU (vCPU), and input/output (I/O) block size, among other factors [27].

Following the above considerations, both environments will require virtual machines able to sustain load from hosted applications, that is, automation server in the case of automation environment, and "Spring Petclinic" with all Kubernetes components in case of target environments. To fulfill these requirements VMs with 2vCPU and 4GB of memory will be created with 16GB disks attached for storing artifacts, dependencies, and different necessary tools or libraries. This choice will also simplify comparison based on the price because virtual machines with these resources are being offered by all the cloud providers, which is not the case for the smallest possible virtual machine.

It is also important to note that some of the components that are needed for both environments are always free, with some quantity limits that go far beyond the needs of this thesis. These are, in most cases, created automatically by cloud providers with the possibility of further customizations. For example, virtual networks or network interface cards (NIC) belong to these resources. As they are always free for each of the cloud providers, they will be omitted in the pricing comparison.

To start comparing cloud providers, the first thing that had to be done is to create an account for each supplier. This process can be slightly different depending on the region. Each country might have different laws, regulations, taxes, etc. Nevertheless, as stated previously end users for this project are in Poland, so this region will be taken into account. For Poland, all steps needed for creating accounts were straightforward and didn't cause issues.

To create a root account, besides typical information like email, password, or phone number, there is a need for giving a credit or debit card number and billing address for two reasons – firstly for authentication purposes, and secondly and more importantly for further billing. What needs to be kept in mind is the fact that this process can be completely different for big organizations when they are setting or migrating solutions to the cloud. Offer, pricing or support might differ depending on each individual contract.

After accounts were set up and root user were logged in the home page of different admin consoles appear. They're presented in Figures 3.3, 3.4, and 3.5 (with personal information like an email address or account name being removed).

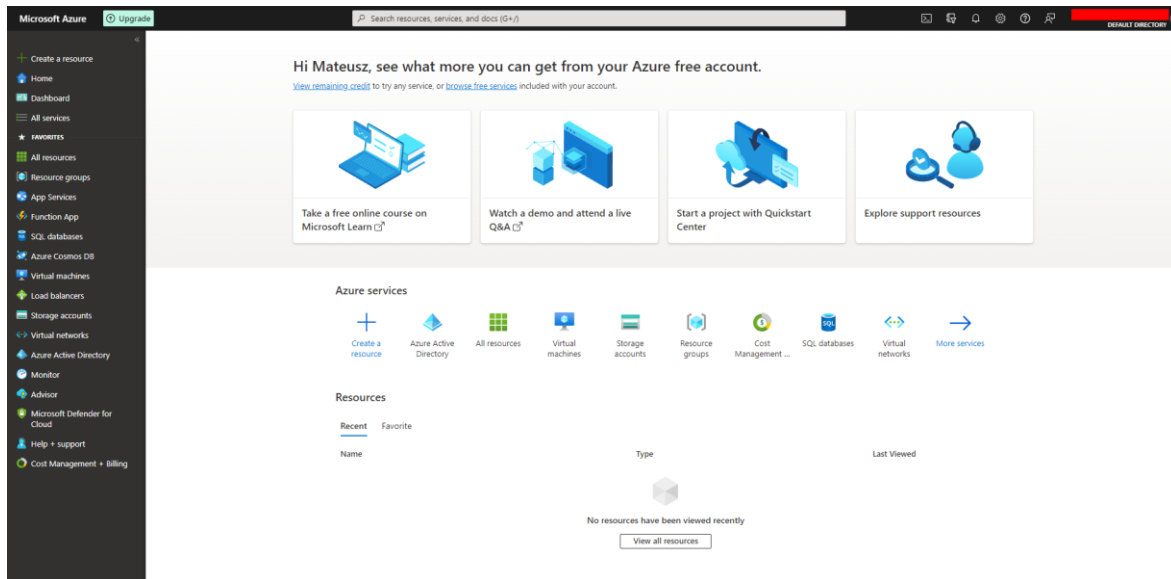


Figure 3.3 Azure Cloud Console – Welcome page

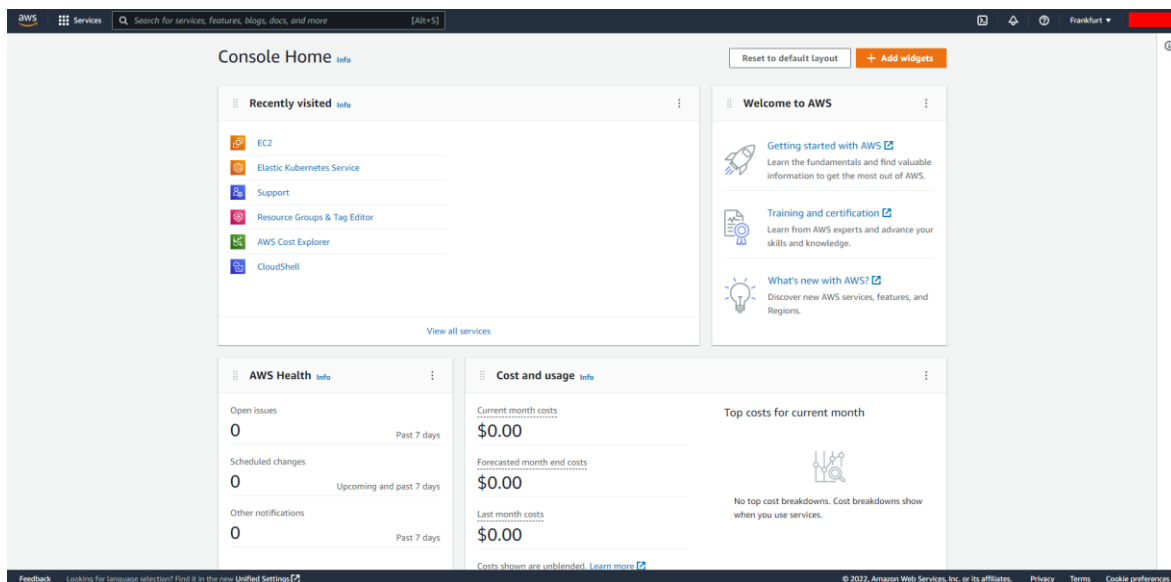


Figure 3.4 AWS Cloud Console – Welcome page

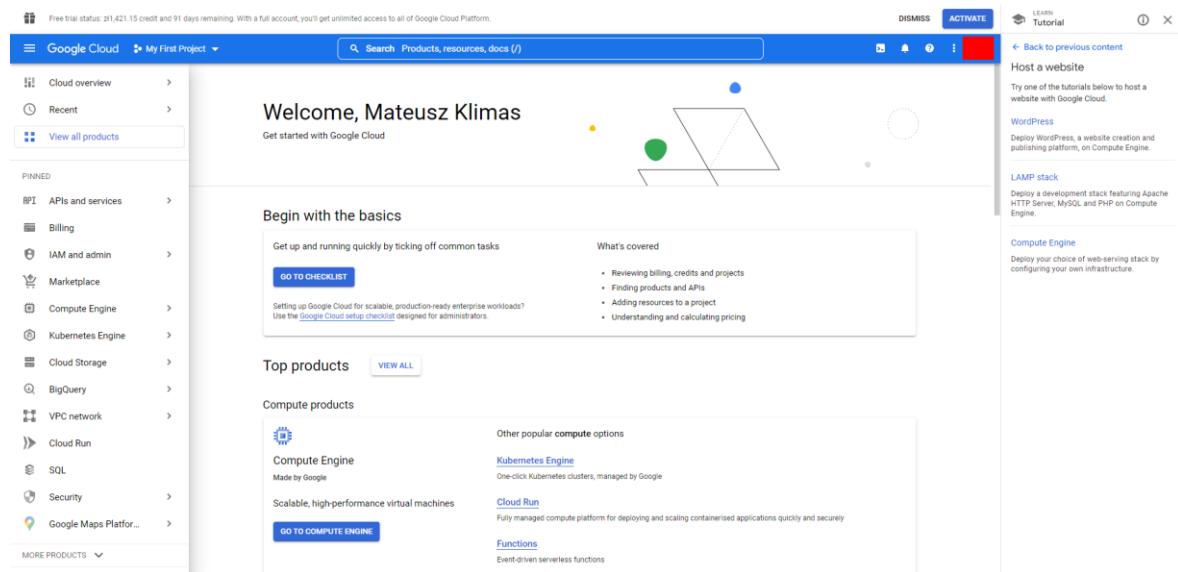


Figure 3.5 GCP Cloud Console – Welcome page

At this point, user can create resources under his account. This leads to the next topic, which is communication with the cloud. Each cloud provider gives different ways of talking to their data centers, but three commons for all providers will be discussed. These are: through the admin user interface, command-line interface, and cloud shell. Admin UI is a graphical interface that can be accessed through a web browser (presented above in Figures 3.3, 3.4, and 3.5). It gives full control over the account and besides managing resources, it can generate graphs for billing, performance, logs, etc. Next, there is a CLI, that must be installed on the user's workstation. These CLI can be installed on either different distributions of Linux, Windows command prompt, or PowerShell. As both, Azure and PowerShell are Microsoft's products, this company offers additional integration between these two.

The last one is a cloud shell which is nothing else than a CLI in a cloud. Nevertheless, it's important to keep in mind that the cloud shell will create some resources, like mounting storage, to be able to function correctly. The most basic versions are free for all of the providers. Of the three, much better Cloud Shells have Azure and GCP. Both, apart from the shell, also offer an integrated editor and file browser. Also, they give a possibility to browse everything in the admin UI without closing the shell. Azure is the only one that provides a choice of shell. User can decide if he wants to use bash or PowerShell, while AWS and GCP have their integrated shells.

When the user selects the most suitable method of deploying resources the next step is to consider the costs related to the usage of cloud computing. Azure, AWS, and GCP offer pay-as-you-go payment option which mean that the user pays only for the resources he uses. Also, each of them gives a pricing calculator which is a tool for estimating costs that a given

environment will generate. These are not perfectly accurate, due to the constantly changing variables, but they give a good estimation of the money that will be spent. All of them also offers discounts on their product, mainly for the reserved resources. For example, Azure offers a ~32% discount if the user reserves VM for the whole year, meaning this VM will be running all the time in that period [31]. For simplicity, discounts won't be taken into account. These calculators work as follows: the user selects services he needs, then he configures their parameters, like the type of the resource or uptime and the estimation is created.

When talking about pricing, it is important to mention that all cloud providers offer free trials, which can be very convenient for everyone who wants to discover the cloud for themselves. Generally, all of the three providers have a similar free tier which is divided into three segments: products that are always free, products that are free for the time of the trial with limitations and trial or some number of credits that can be spent however the user wants [33],[34],[35]. To put it into perspective: for their trial, Azure offers 40+ services that are always free, popular services with limitations for 12 months, for example, 750 hours of B1s (1CPU, 1GB memory) Azure Virtual Machines each month or 250Gb SQL Database and 200 USD in credits that can be used in 30 days after activation [33]. Apart from that each of the cloud providers has an alternate offer for startups or small businesses.

Estimations for each cloud provider are made in chapter 5.1. It's important to note, that when creating estimations for the environments, different nuances must be reviewed, like:

- Prices of resources differ from the region, but since latency is somewhat important all the resources will be located in West Europe or Frankfurt.
- In opposition to Azure and GCP, in Amazon user pays for static IP addresses only when they are not used e.g., the situation when virtual machines that have assigned IP address to them are turned off.
- Assumption is that all VMs must be available all the time and since dynamic IP addresses are free for all providers, there is a possibility to skip static IP addresses, but there can be situations where the VM will have to be turned off and after that operation, when the VM will be turned on again, a new IP address will be assigned

After resources are selected, the task of choosing the most appropriate data center must be carried out. The client's request processing time by the server depends on many different variables. Among them, an important one is the physical distance between the client and the server. This is the reason why companies build their data centers in different locations, so the server is as close to the end user as possible to provide a faster connection. This scenario is no different for cloud providers. Their customers are spread around the world, and to provide reliable services, their data centers are built across many locations. Besides that,

different data centers often come with different costs, because the price of maintaining a data center is not the same depending on the region.

When talking about cloud data centers there are two very important concepts. These are regions and availability zones. Each region is a separate geographical area and availability zones are isolated, physically separated locations within each region [35]. Therefore, regions can be treated as collections of zones. Zones located in the same region are connected with the usage of high-bandwidth and low-latency network, providing the lowest possible latency. Also, availability zones have their own, separate power, cooling, and networking infrastructure. The purpose of availability zones is to ensure that even if one zone is compromised, the other two can still provide regional services, capacity, and high availability [36]. This approach protects the services from software or hardware failures or events like fires, floods, or earthquakes. To ensure even higher availability or prevention against large-scale disasters that can damage the whole region, data can be replicated in another region with disaster recovery mechanisms. The scheme of such architecture is presented in Figure 3.6.

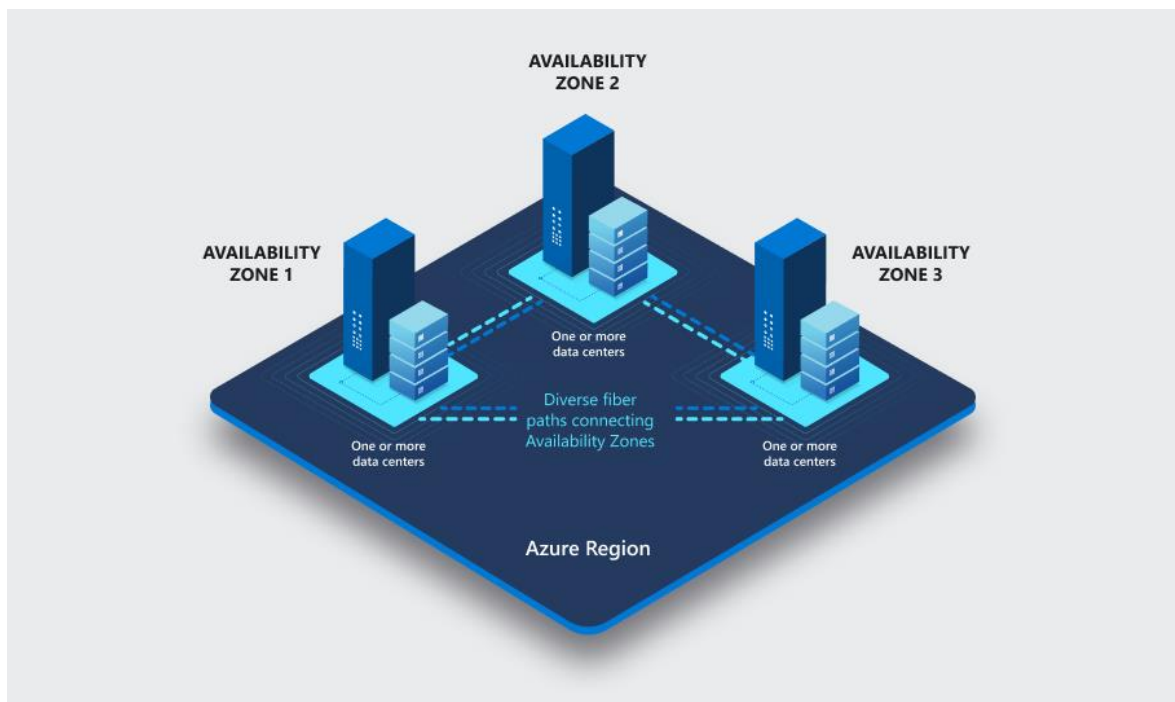


Figure 3.6 Availability zones in Azure,

Source: <https://learn.microsoft.com/en-us/azure/availability-zones/az-overview>

Choosing the appropriate region can drastically change the end user experience and may decrease or increase costs. Latency is an important aspect in the user's experience. It is the amount of time it takes for data to go from one place to another. It can be measured in two ways. These are round trip time (RTT) and time to first byte (TTFB). The first one is the amount of time the packet needs to get from the client to the server, and then, from the server

to the client. TTFB is the time that the server needs to receive the first byte of the request's data.

To measure RTT and TTFB two Linux commands can be used – *ping* and *curl*, where the second one stands for “client URL”. *Ping* operates by sending one or more Internet Control Message Protocol (ICMP) Echo Request packets to a predetermined target IP on the network and waits for a response. The destination sends back an ICMP echo reply after receiving the packet. After that round-trip time is calculated. The syntax for *ping* command is as follows:

ping [OPTIONS] DESTINATION

Example ping output is presented in Figure 3.7. By default, after invoking this command it will run until the user stops the execution. To specify the number of times the *ping* is invoked, there is a need to add the *-c* option which stands for the count. Upon finishing, the *ping* command summarizes its results and prints them to the user.

```

~ $ ping google.com
PING google.com (216.58.215.78) 56(84) bytes of data.
64 bytes from waw02s16-in-f14.1e100.net (216.58.215.78): icmp_seq=1 ttl=56 time=30.7 ms
64 bytes from waw02s16-in-f14.1e100.net (216.58.215.78): icmp_seq=2 ttl=56 time=21.9 ms
64 bytes from waw02s16-in-f14.1e100.net (216.58.215.78): icmp_seq=3 ttl=56 time=26.3 ms
64 bytes from waw02s16-in-f14.1e100.net (216.58.215.78): icmp_seq=4 ttl=56 time=20.4 ms
64 bytes from waw02s16-in-f14.1e100.net (216.58.215.78): icmp_seq=5 ttl=56 time=39.6 ms
64 bytes from waw02s16-in-f14.1e100.net (216.58.215.78): icmp_seq=6 ttl=56 time=23.4 ms
^C
--- google.com ping statistics ---
6 packets transmitted, 6 received, 0% packet loss, time 5008ms
rtt min/avg/max/mdev = 20.361/27.039/39.627/6.549 ms

```

Figure 3.7 Example output of *ping* command

Curl, on the other hand, is a utility for transferring data from the server or the other way around. It is capable of doing much more than a *ping* command. Its syntax is the same as in the case of *ping*:

curl [OPTIONS] DESTINATION

To execute a command the specified number of times, it will be encapsulated into *for* command. The full command is presented in Figure 3.8 and the example output is presented in Figure 3.9. All options are used for output formatting purposes.

```

1 for i in {1..100};do curl -o /dev/null -H 'Cache-Control: no-cache' -s -
  w "TTFB: ${time_starttransfer} \n" https://www.google.com/; done

```

Figure 3.8 *Curl* command used in the experiment described in chapter 5.2

```
matthew@DESKTOP-BFVML8D:~$ for i in {1..5};do curl -o /dev/null -H 'Cache-Control: no-cache' -s -w "TTFB: %{time_starttransfer} \n" https://www.google.com/; done
TTFB: 0.486173
TTFB: 0.479603
TTFB: 0.439248
TTFB: 0.438696
TTFB: 0.443191
```

Figure 3.9 Example output of curl shown in Figure 3.8

To demonstrate how regions can influence latency and pricing, there was an experiment, described in chapter 5.2, performed. Results from the mentioned test helped choose a cloud provider in further chapters. This leads to another topic, which is choosing appropriate CI/CD tools, which is discussed in the next chapter.

3.3. Build Tools

Each major programming language has its build automation tools. Their main task is to generate build artifacts by performing actions, like compiling and linking source code. Apart from that, many build tools also have capabilities of dependency management. For Java-based projects, the most popular build tools are Apache ANT, which is an abbreviation of “Another Neat Tool”, Apache Maven, and Gradle. All three of them are released under Apache License 2.0. In this subchapter, provided will be a brief introduction to all of them, and then the best tools will be chosen or discarded for further tests.

Apache Ant was the first build tool made for Java-based applications. Initially, made to replace Make, which in early Java days was the only available build tool. Apart from building Java projects, ant has capabilities of building apps written in different languages. Ant originates from Apache Tomcat and was released on 19th July of 2000 as a standalone project.

Ant build files are written in Extensible Markup Language (XML), usually called build.xml. Example Ant build.xml file is presented in Figure 3.10 Ant separates build process into phases which in the XML file are called targets. In the presented snippet there are defined four targets that depend on each other. That means that if the compile phase will be invoked it will first run a clean target [37].

```
1 <project>
2   <target name="clean">
3     <delete dir="classes" />
4   </target>
5
6   <target name="compile" depends="clean">
7     <mkdir dir="classes" />
8     <javac srcdir="src" destdir="classes" />
9   </target>
10
```

```

11     <target name="jar" depends="compile">
12         <mkdir dir="jar" />
13         <jar destfile="jar/HelloWorld.jar" basedir="classes">
14             <manifest>
15                 <attribute name="Main-Class"
16                     value="antExample.HelloWorld" />
17             </manifest>
18         </jar>
19     </target>
20
21     <target name="run" depends="jar">
22         <java jar="jar/HelloWorld.jar" fork="true" />
23     </target>
24 </project>

```

Figure 3.10 Example Ant build.xml file

Ant is regarded as one of the most flexible build tools. It does not specify any conventions in the code or project structure. In that scenario, developers are required to write all the commands. This can lead to enormous XML files, which in the end can be hard to maintain. This is a big drawback for Ant because while joining the project as a new member of the team, learning a complicated build.xml file can be time-consuming. Also, initially Ant, by default, did not have the possibility of dependency management, which later was solved by a project called Apache Ivy which now is integrated with Ant.

Four years later, on 13th July of 2004, the Apache Maven was released. Like Ant, Maven also uses XML for configuration files, but considering the build.xml complexity problem, it is doing it in a more user-friendly fashion. In opposition to Ant, Maven is based on conventions and comes with already defined commands, which are called goals. These are equivalent to targets in Ant. Instead of writing everything from the scratch, Maven provides a framework for building the project. In addition to it, Maven by default provides dependency management for the project.

The Maven configuration file is called pom.xml, which stands for Project Object Model. It stores build instructions as well as dependency management or project properties [38]. An example of such a file is presented in Figure 3.11. Maven's configuration file and project structure are standardized. Another difference that can be noticed, compared to Ant, is the fact that no build phases were defined. Instead, Maven provides built-in functions.

```

1 <project xmlns="http://maven.apache.org/POM/4.0.0"
2     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
4         http://maven.apache.org/xsd/maven-4.0.0.xsd">
5     <modelVersion>4.0.0</modelVersion>
6     <groupId>baeldung</groupId>
7     <artifactId>mavenExample</artifactId>
8     <version>0.0.1-SNAPSHOT</version>
9     <description>Maven example</description>
10
11     <dependencies>

```



```
12     <dependency>
13         <groupId>junit</groupId>
14         <artifactId>junit</artifactId>
15         <version>4.12</version>
16         <scope>test</scope>
17     </dependency>
18 </dependencies>
19 </project>
```

Figure 3.11 *Example Maven pom.xml file*

Going further, Maven provides a default build lifecycle. Each of the goals can be executed separately, remembering that all previous ones will be run as well. The goals are:

- validate,
- compile,
- test,
- package,
- verify,
- install,
- deploy [39].

Since all tasks are performed by plugins, Maven may be thought of as a framework for the execution of plugins, as stated in the official documentation [40]. A large variety of plugins are supported by Maven, and each one may be further customized.

All the improvements over Ant made Maven very popular among Java developers. Standardized configuration file and project structure made it much more manageable compared to Ant. On the other hand, all of these strict conventions significantly decrease the flexibility of Maven. The creators of the next discussed build tool tried to combine what's good from Ant – flexibility with Maven features.

Gradle, released in 2008 was built on top of the ideas that stand behind Ant and Maven. The biggest change compared to its predecessors is that Gradle is not using XML files. Instead, it uses domain-specific language (DSL). It is a language that allows solving problems for the specialized application domain. Gradle gives developers a choice between Groovy or Kotlin-based DSL. The main configuration file Gradle uses is called `build.gradle` in Groovy (or `build.gradle.kts` for Kotlin) [41]. An example of such file is presented in Figure 3.12.

```
1  apply plugin: 'java'
2
3  repositories {
4      mavenCentral()
5  }
```

```
6
7 jar {
8     baseName = 'gradleExample'
9     version = '0.0.1-SNAPSHOT'
10 }
11
12 dependencies {
13     testImplementation 'junit:junit:4.12'
14 }
```

Figure 3.12 *Example Gradle build.gradle file*

Following Maven’s plugin idea, clean installation of Gradle comes with very few features. Plugins are essential for Gradle’s functionalities. Looking at the first line of the example, plugin java is being applied. It allows compiling code written in Java, besides its other capabilities. For running spring boot-based Java applications, such as “Spring PetClinic”, the “org.springframework.boot” is used.

While analyzing brief introductions of the three build tools, given in this subchapter, Gradle should be the choice for every Java project. Nevertheless, based on the “State of Java” survey [42] in 2016 Maven was used in 72.5% of Java projects and Gradle in 19%. One year later the same survey showed that Maven was used in 76% and Gradle only in 18%. According to the newer report called “2021 Java Developer Productivity Report” 67% of developers were using Maven as a build tool, 20% Gradle, and 11% used Ant [43]. These surveys show that despite all of Gradle’s advantages, some developers still prefer, a more mature tool, which is Maven.

Regarding performance, Gradle claims that their product is up to 100 times faster than Maven [44]. They run many different builds on different-sized projects using the same machine. In every single scenario, Gradle was at least 2 times faster than Maven. They also shared every needed resource to reproduce their results.

Taking into consideration the fact that Maven is the most popular build tool, while Gradle is the newest and most modern solution, both will be tested in further read. A very important aspect that was not mentioned and will be analyzed is compatibility with different tools. To examine integrity with automation servers, performance based on “Spring PetClinic” and usage of resources both build tools will be used in different configurations with automation servers, which will be the next topic of the thesis.

3.4. Automation Servers

Automation servers often called “CI/CD tools” are there to help developers and operation teams to build, test, deploy and, many more. There are many automation servers from open-

source projects to expensive enterprise solutions. There are ones that are hosted by the providers or by the users. In this, chapter tools will be briefly compared, then, the choice will be narrowed to two, and finally, based on the tests, one of them will be chosen for the final project.

Whenever looking for the best or the most popular tools, a few names are always appearing. These are Jenkins, CircleCI, Bamboo, GitLab CI, and TeamCity among the other solutions. Apart from them, Azure, AWS and GCP are also offering their solutions that help implement CI/CD. These are respectively called: Azure Pipelines, AWS CodePipeline, and Cloud Build. Since Azure was chosen as a cloud provider for the project, only Azure Pipelines will be compared with other available solutions.

As previously to knowingly chose a proper solution, it is important to specify the requirements for the project. Prices for the CI/CD tools are evaluated differently from the ones in previous chapters. The main factors defining the price are CI/CD, or build, minutes per month, and build agents or in some cases the number of builds that can be run in parallel. The first one specifies how many minutes monthly a user can run his builds and concerns only solutions that are hosted by the provider of a given solution. Build agents are programs that are deployed on the node and are then instructed by a controller to perform a job. One agent can run multiple parallel builds, depending on the number of cores, a given node has. This metric concerns solutions hosted by the user.

The first reviewed aspect will be hosting the solution. Jenkins is the only open-source solution on this list, meaning, it must be hosted by the user. Another solution that allows using their application for free when hosted by the user is TeamCity with some limitations, that is up to 100 build configurations and three build agents, which is more than enough for the project. Azure Pipelines offers both options in free and paid versions. CircleCI allows hosting their solution by the user, but to this, the user needs to contact them to build a custom solution. The rest is hosted only by the provider.

The next consideration will be the pricing of the solution. Jenkins and TeamCity can be used for free in case of installing software on the user's infrastructure. Then, providers usually have a free plan that includes some number of minutes per month, build agents, or builds in parallel. An exception here is Bamboo, which offers a 30-day trial only, and TeamCity, which besides allowing the use of their solution, has only a 14-day trial in case of hosting application by them. Then, there are paid plans usually for smaller and bigger organizations. Since this project doesn't require much build time and is assumed to consume as little money as possible, firstly, free plans will be reviewed. These in addition to the basic information mentioned above are presented in Table 3.1. Keeping in mind that a self-hosted, single virtual machine will cost 12.09 USD if run 24 hours each day, which gives 45,750 minutes of builds per month (assuming that on average month will have 30.5 days).

Table 3.1 *Overview of the selected features of the automation servers*

Solution / Feature	Open-source	Hosted By	Free min/month	Number of free agents
Jenkins	Yes	User	Unlimited	Unlimited
CircleCI	No	User / provider	6000	Not specified
Bamboo	No	Provider	30-day trial	0
GitLab CI	No	Provider	400	Not specified
TeamCity	No	User / provider	Unlimited / 14-day trial	3
Azure Pipelines	No	User / provider	Unlimited / 1800	1

Besides Bamboo, which pricing starts at 1200 USD annually for 1 agent, all other solutions have their pros and cons and there is no clear winner here. Two solutions will be compared, one being hosted purely by the user and one by the provider. Based on its popularity and the fact that it is open-source, Jenkins will be the first technology reviewed further. As per second choice, it will be CircleCI. The main reason behind this choice is simply its affordable price and the best free plan.

Selected automation servers will be discussed in more detail in the next chapter to help establish the more suitable solution. Additionally, with a combination of build tools – Gradle and Maven – an experiment was performed, which is described in chapter 5.3. Along with all the research, it will help to determine the tools used in the final implementation.

3.4.1. Jenkins

Jenkins is an open-source automation server, which helps achieve goals assumed by DevOps methodology. It's also by far the most popular solution on the market. It is written in Java programming languages and had its debut on 2 February 2011. It is released under the MIT License, which gives users rights to using, copying, or modifying original or alternated application. Jenkins has two release lines, which are Long-Term Support (LTS) and weekly releases. The first one, being more stable, is meant for production systems, while the second one delivered more frequently with bug fixes is usually meant for development systems or experiments [45]. As this project won't go that deep into Jenkins nuances to expose any issues with any of the releases, it does not matter which release line will be used. For this work, Jenkins 2.346.3 LTS will be used.

Jenkins can be used as a generic Java package (.war extension), as a container or service on different Linux distributions, macOS or Windows. Usually, it runs as a standalone application with the help of a built-in application server, called Jetty, which can be an alternative for a Tomcat. Jenkins architecture is presented in Figure 3.13. Jenkins by default

communicates with the user or another integration APIs with either HTTP over port 8080, or HTTPS over port 443.

After installing Jenkins, exposing port 8080 and starting Jenkins service, its UI is available under <virtual-machine-IP>:8080 typed in the browser. After the first start, Jenkins needs to be unlocked with the key stored on the machine, where installation was made (this key is also present in logs after starting for the first time). Like Gradle, standalone Jenkins installation doesn't provide a lot of utilities and the key to its functionality are plugins. Even during the installation process Jenkins asks the user what plugins should be installed. It also gives option of installing "suggested plugins", which for example includes plugins for communicating with git, SSH, and the most popular build tools, like Gradle, etc. Despite installing plugins, to use for example git command, git VSC must be installed on the virtual machine, since plugins are only the communication with the underlying operating system and programs. After creating an administrator user, Jenkins home page is displayed to the user which is presented in Figure 3.13.

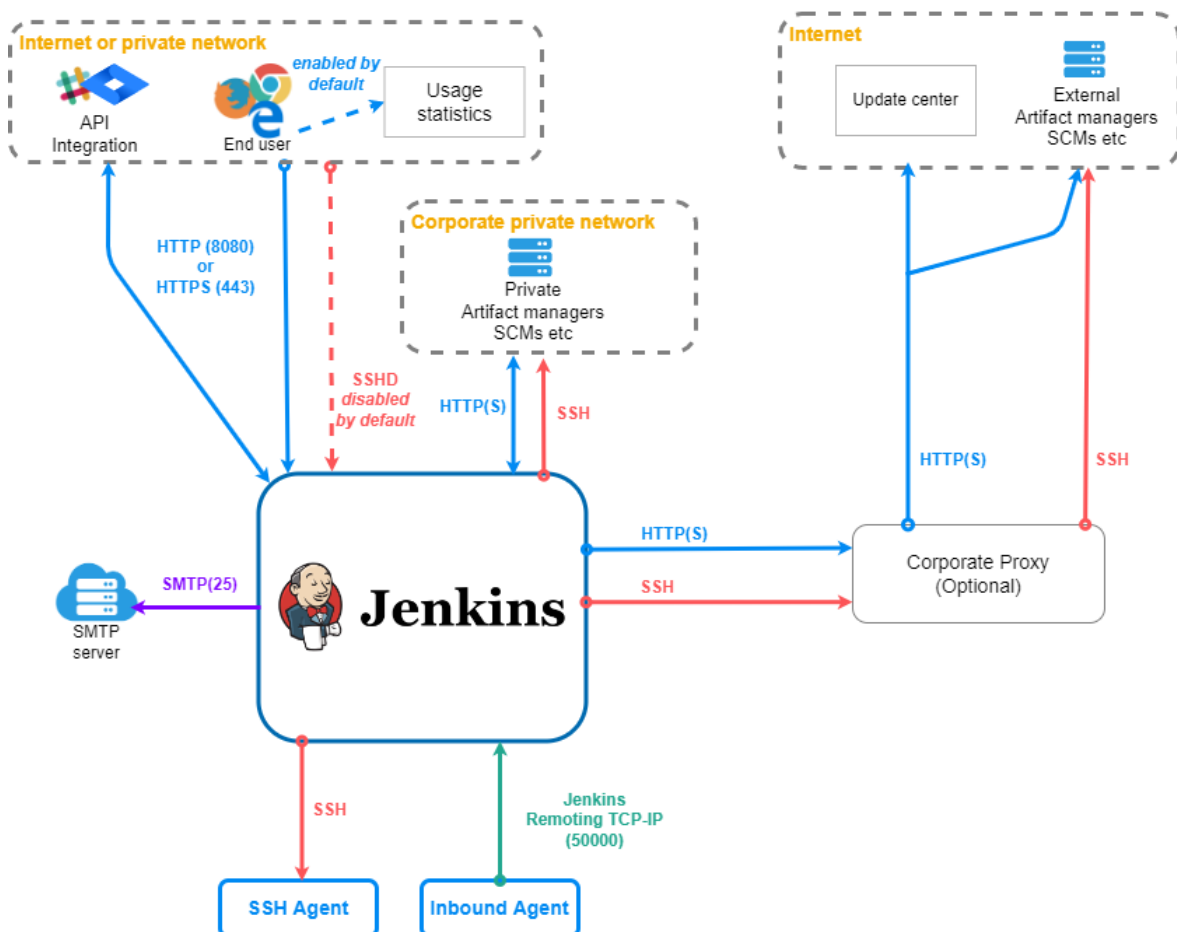
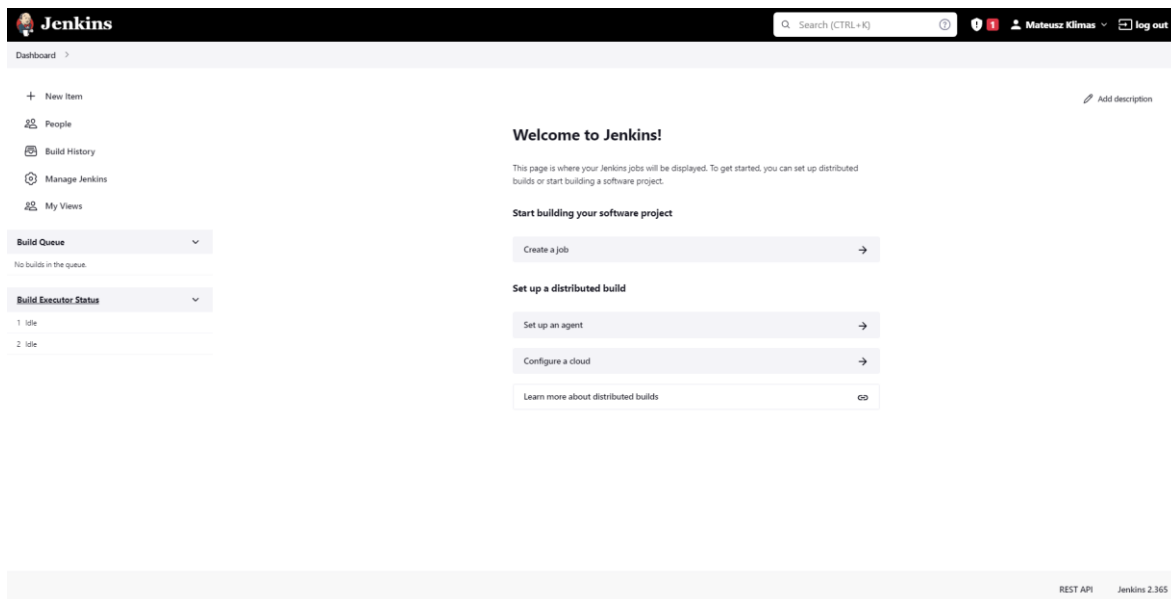


Figure 3.13 *Jenkins architecture*,
Source: <https://www.jenkins.io/doc/developer/architecture/>

Figure 3.14 *Jenkins Home Page*

The default way of creating a Jenkins project is called “Freestyle project”, where the user defines everything in the browser window and then the configuration is being stored under the project’s directory in a file named `config.xml`. With the addition of many plugins, this approach can quickly become inconvenient when the user must scroll up and down to properly configure the project. This was the only way of creating projects in Jenkins 1.x and was criticized by the community.

These issues led to introducing Jenkins Pipeline functionality in version 2.x. It is nothing else than a set of plugins that supports implementing and integrating continuous delivery pipelines into Jenkins [46]. The definition of such pipeline is stored as a text file called `Jenkinsfile`. Such an approach allows putting the file directly into the project’s repository, enabling an approach called “Pipeline-as-code”. This comes with many benefits such as the repository being the single source of truth, allowing everyone to view such files. Also, it allows Jenkins to download `Jenkinsfile` from the repository and run it, instead of storing it on the machine running it [46].

`Jenkinsfile` can be written in two different ways. First of which is called scripted pipeline, which was the first version of the “pipeline-as-code”. Its syntax is built on top of the Groovy, providing great flexibility, and allowing it to inject code into the pipeline. Nevertheless, it requires some knowledge of Groovy, which can be problematic. Besides that, due to possibility of injecting the code `Jenkinsfile` could get complicated. An example of the scripted pipeline is presented in Figure 3.15.

```
1 node {
2     stage('Test') {
3         git 'https://github.com/user/repository.git'
4         sh 'mvn test'
5         archiveArtifacts artifacts: 'target/surefire-reports/**'
6     }
7     stage('Build') {
8         sh 'mvn clean package -DskipTests'
9         archiveArtifacts artifacts: 'target/*.jar'
10    }
11    stage('Deploy') {
12        sh <perform-deploy>
13    }
14 }
```

Figure 3.15 *Example of scripted pipeline in Jenkins*

An alternative and more recent approach that will be used in the project is declarative syntax. The equivalent of the previous pipeline written in declarative syntax is shown in Figure 3.16. It is composed of the five main components:

- pipeline – contains the content of the pipeline,
- agent – defines the machine that will run the pipeline,
- stages – defines pipeline's stages,
- stage – defines subset of tasks performed through the Pipeline e.g. 'build',
- steps – defines a single task.

Another difference is that the declarative pipelines contain directives. They are used to add additional logic to the pipeline. It helps set properties, like triggers or environment variables. Stages by default are executed from top to bottom.

```
1 pipeline {
2     agent any
3     tools {
4         maven 'maven'
5     }
6     stages {
7         stage('Test') {
8             steps {
9                 git 'https://github.com/user/project.git'
10                sh 'mvn test'
11                archiveArtifacts artifacts: 'target/surefire-reports/**'
12            }
13        }
14        stage('Build') {
15            steps {
16                sh 'mvn clean package -DskipTests'
17                archiveArtifacts artifacts: 'target/*.jar'
18            }
19        }
20        stage('Deploy') {
21            steps {
```

```

22         sh 'echo Deploy'
23     }
24 }
25 }
26 }

```

Figure 3.16 *Example of declarative pipeline in Jenkins*

After knowing most the basic Jenkins features, the next chapter will focus on its competitor, which is CircleCI.

3.4.2. CircleCI

CircleCI is a CI/CD platform from a software company called Circle Internet Services, Incorporated. The company was founded in September 2011 and after that its flag product quickly became one of the most popular CI/CD tools. In contrast to Jenkins, CircleCI is managed by the software provider, but it is also possible to use their product hosted by the user.

CircleCI offers four different plans for using their platform Free, Performance, Scale, and Server. The last option is meant for projects hosted by the user and for this option there is a need for contacting CircleCI for designing such infrastructure [47]. The rest of the plans offer the following specifications. Starting from the Free plan:

- Up to 6000 minutes per month.
- Possibility of choosing the right resource class (S-L), these refer to containers that will run the jobs.
- Up to 30 parallel jobs.

The next one is Performance plan, which comes at the price of 15 USD per month:

- 6000 bonus build minutes per month.
- 5 users, and 15 USD per additional user.
- Access to additional resource classes.
- Up to 80 parallel jobs.
- Access to support available 8 hours each working day.

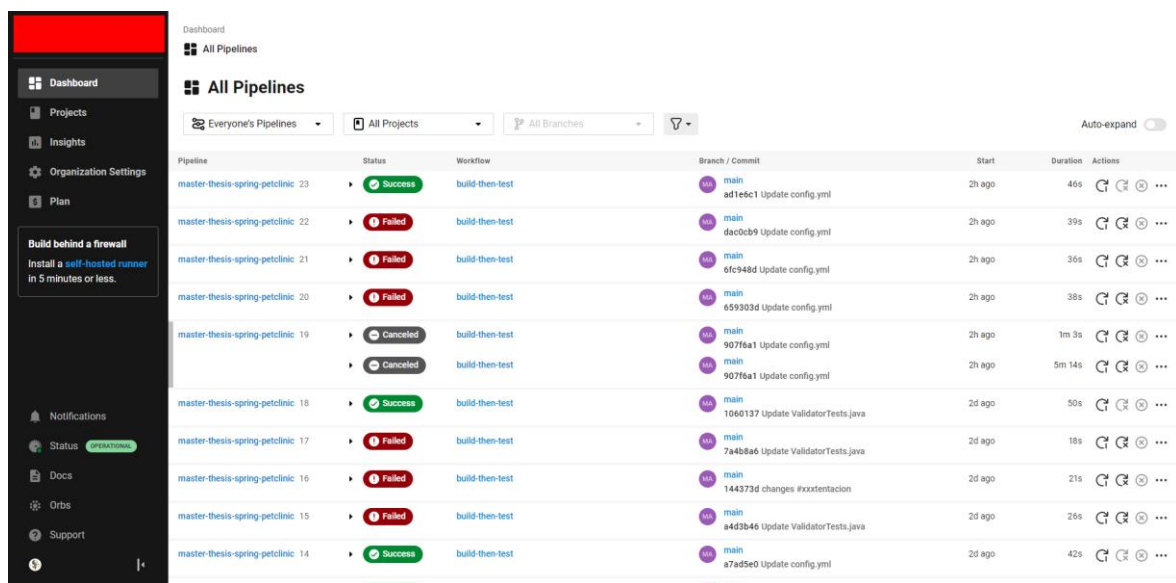
The last plan is Scale with the price of 2000 USD per month:

- Ability to customize build minutes and user's seats.
- All resource classes with the addition of GPU resource classes.
- Access to support available at any time.

- Customizable annual billing, audit logging, and bulk data export [47].

Looking at the prices and comparing them to the competition CircleCI offers affordable options for its customers.

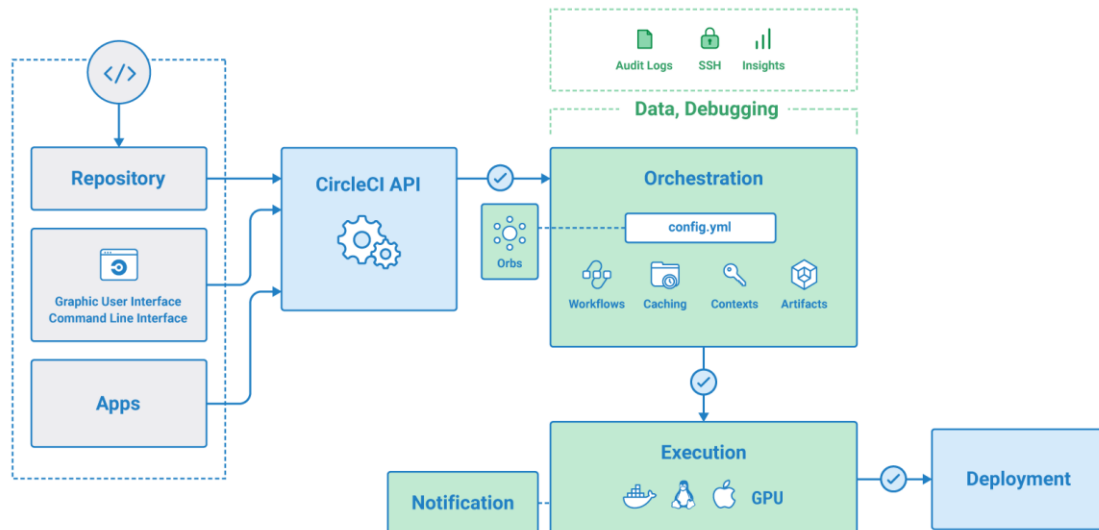
CircleCI works as a web application, thus it is controlled via a web browser. An example view of the application's dashboard is shown in Figure 3.17. After creating an account, the user is asked to connect his VCS, in this case it is GitHub. After that, all repositories are listed and CircleCI gives an option to follow the project which will make builds related to this repository appear on the dashboard. For configuring CircleCI, a directory called ".circleci" must be created inside the repository with the file "config.yml" inside. This is the bare minimum for making the project compatible with the platform.



Pipeline	Status	Workflow	Branch / Commit	Start	Duration	Actions
master-thesis-spring-petclinic: 23	Success	build-then-test	main ad1e6c1 Update config.yml	2h ago	46s	🔄 📄 ⚙️ ⋮
master-thesis-spring-petclinic: 22	Failed	build-then-test	main dac0cb9 Update config.yml	2h ago	39s	🔄 📄 ⚙️ ⋮
master-thesis-spring-petclinic: 21	Failed	build-then-test	main 6fc948d Update config.yml	2h ago	36s	🔄 📄 ⚙️ ⋮
master-thesis-spring-petclinic: 20	Failed	build-then-test	main 659303d Update config.yml	2h ago	38s	🔄 📄 ⚙️ ⋮
master-thesis-spring-petclinic: 19	Canceled	build-then-test	main 907f6a1 Update config.yml	2h ago	1m 3s	🔄 📄 ⚙️ ⋮
master-thesis-spring-petclinic: 19	Canceled	build-then-test	main 907f6a1 Update config.yml	2h ago	5m 14s	🔄 📄 ⚙️ ⋮
master-thesis-spring-petclinic: 18	Success	build-then-test	main 1060137 Update ValidatorTests.java	2d ago	50s	🔄 📄 ⚙️ ⋮
master-thesis-spring-petclinic: 17	Failed	build-then-test	main 7a4b8a6 Update ValidatorTests.java	2d ago	18s	🔄 📄 ⚙️ ⋮
master-thesis-spring-petclinic: 16	Failed	build-then-test	main 144373d changes #xxentacion	2d ago	21s	🔄 📄 ⚙️ ⋮
master-thesis-spring-petclinic: 15	Failed	build-then-test	main a4d3b46 Update ValidatorTests.java	2d ago	26s	🔄 📄 ⚙️ ⋮
master-thesis-spring-petclinic: 14	Success	build-then-test	main a7ad5e0 Update config.yml	2d ago	42s	🔄 📄 ⚙️ ⋮

Figure 3.17 CircleCI dashboard

Apart from the manual way of running the jobs, by default, each commit to the repository will trigger the job associated with the repository. Each job is being run by a CircleCI in a separate container or a virtual machine, which can be specified in the configuration file [48]. The architecture of CircleCI is shown in Figure 3.18. In contrast to Jenkins CircleCI doesn't rely on plug-ins. Most of its functionality works out-of-the-box (OOTB) and doesn't require any additional installations.

Figure 3.18 *CircleCI Architecture*

As per configuration file, its structure is very similar to the previously discussed Jenkins file. The example configuration file is shown in Figure 3.19. First line of the configuration file must be version of the CircleCI server. Then, usually file is split into two sections. The first is the jobs section where the user defines jobs that needs to be run. As CircleCI runs each job on a separate container or VM, this property is mandatory. In this case, it is a lightweight container with Java support. Each of the jobs has its own steps which are then listed in the jobs overview from CircleCI UI and can be further examined or troubleshot. In the end, the user defines the workflow for the project. From the example in Figure 3.19, two jobs are defined. The first one, called the “*build*” job has to be executed first. Then, after successful execution, the “*test*” job will run. In the case when the “*build*” job failed, the “*test*” will be skipped. By default, the “*workflows*” section will be executed from the top to bottom, unless specified otherwise.

```

1  version: 2.1
2
3  jobs:
4    build:
5      docker:
6        - image: cimg/openjdk:17.0.1
7      steps:
8        - checkout
9        - run: echo "this is the build job"
10   test:
11     docker:
12       - image: cimg/openjdk:17.0.1
13     steps:
14       - checkout
15       - run: echo "this is the test job"
16
17   workflows:

```

```
18  build_and_test:
19    jobs:
20      - build
21      - test:
22          requires:
23            - build
24
```

Figure 3.19 *Example pipeline in CircleCI*

An important aspect of working with CircleCI is Orbs. These are reusable snippets of the code accelerating setting up the project, integrating with other tools, and more. Also, they greatly reduce the complexity of the configuration file. Examples of such can be jobs, commands, or executors, which are definitions of containers or virtual machines, the job will be run on. Orbs can be found in the open repository from CircleCI. Each user can create his own public or private orbs to simplify and speed up his project [49].

3.5. Summary

After comparing Azure, AWS, and GCP, based on the experiment described in chapter 5.1, it can be concluded that for projects that consist of basic components, like Virtual Machines, disks, IP Addresses, etc., their offer and pricing are very similar. Some nuances can be spotted, but generally speaking, there were little differences. However, larger differences occur while looking at more specific solutions. Even for technology that is really common in today's cloud-like managed k8s cluster, change in pricing is significant. The more specific solution, the bigger differences can be. This example proves that it is worth spending time for at least brief research and calculations before implementation of the project. This also gives an overall view of offered resources and working with each of the providers.

As shown during examining latency, in experiment from chapter 5.2, regions also play an important role. If most of the end users are located in Europe to provide faster connection, it is crucial to deploy resources onto a data center that's close to them. In this test, GCP had the best results, since the data center in Warsaw is closest to the workstation from which requests were made. Nevertheless, the difference between Warsaw and Frankfurt data centers was so similar that it won't be noticeable.

Also, not all resources can be created in every region. For example, the virtual machine of the size t2.micro from AWS is not available in all regions. This applies not only to the specific VM type but also to whole solutions like Google's Artificial Intelligence API. Another factor, that needs to be considered is the fact that not all regions offer the same availability zones, so if the project requires this option, it is important to know about it.

As per ease of use, working with every provider was straightforward. Each admin UI console is well-designed and often guides user when performing actions. Much information presented in the browser has boxes near them, with brief explanations after expanding them. Documentation from each of them is well-written and most of the information can be found very quickly. Also, all providers offer detailed tutorials about the usage of their products. The fact that AWS is the most popular cloud provider causes the situation in which, when looking for a solution to specific problem in general, most results were directly describing how to solve it for AWS. On the other hand, naming conventions and principles of working with the cloud are often shared by the cloud providers. This might allow finding answers for all the cloud providers, even if they're written specifically for one of them.

With all that being said, Azure will be the choice as a cloud provider for the project in this work. This choice is backed mainly by the price of AKS and the fact that Azure allows creation of unlimited free clusters. Besides that, the ease of use is a very subjective thing. Nonetheless, the author has on-hands experience with Azure. This factor, which is experience of the engineers working with the given technology, can play an important role in real-life projects since it can be easier to implement the solution.

Regarding CI/CD tools, based on the experiment in chapter 5.3, Gradle outperformed Maven in every aspect, but it consumed around 20% more memory than Maven. When looking at the project lifecycle Gradle and Maven provides similar possibilities. Both solutions also are well integrated with both CI/CD automation servers. Another big difference can be seen in the configuration file. Gradle's file for this project has exactly 35 lines of text excluding empty lines, while the equivalent for Maven has 101. Gradle's configuration files are more concise and require fewer elements than XML files. Based on these results, Gradle will be the choice for the final pipeline.

After analyzing CircleCI and Jenkins in the context of performance Jenkins was faster even on smaller resources. It can be argued that for bigger projects these changes are not relevant at all, but monolithic applications are not that popular right now and microservices took their place. In the situation where project consists of a bigger number of microservices, this difference will be notable.

A big advantage of CircleCI is that everything works out of the box, and minimal configuration is required. On the other hand, this comes with the price since the user doesn't have to worry about managing any underlying infrastructure. With the usage of concept known as Infrastructure as a Code virtual machine can be provisioned with Jenkins with all necessary plugins automatically, but it still requires some work in opposition to CircleCI. In the end, the user must decide if more control over the system is more important than the lack of the installation and configuring the solution.

While working with both solutions, both UIs were well structured and transparent. Navigation was straightforward and went without issues, but as in the case with every solution, it took a little time to get used to them. Since Jenkins is an open-source solution, its UI looks “older” than commercial CircleCI. While this is purely subjective, Jenkins via plugins gives possibilities of customizing its look, while CircleCI does not.

Another advantage for Jenkins is its big community. Many issues or challenges were already faced and well described. CircleCI community is not that big, thus user has to rely mainly on documentation, which is well written for both solutions. On the other hand, in case of paid plans, CircleCI provides support from their side, which is not the case for Jenkins.

Taking into consideration all these thoughts, Jenkins will be the choice for the project. It is mainly backed by the fact that the technology is open source, and its community makes it much easier while solving issues or trying to accomplish tasks. Another factor is that Jenkins has a very wide library of plug-ins that support a lot of modern technologies. Apart from that in performed tests, Jenkins did better than CircleCI.

With all the performed preparations project can be implemented. Such implementation, with a detailed explanation of all the taken steps will be the topic for the next chapter.

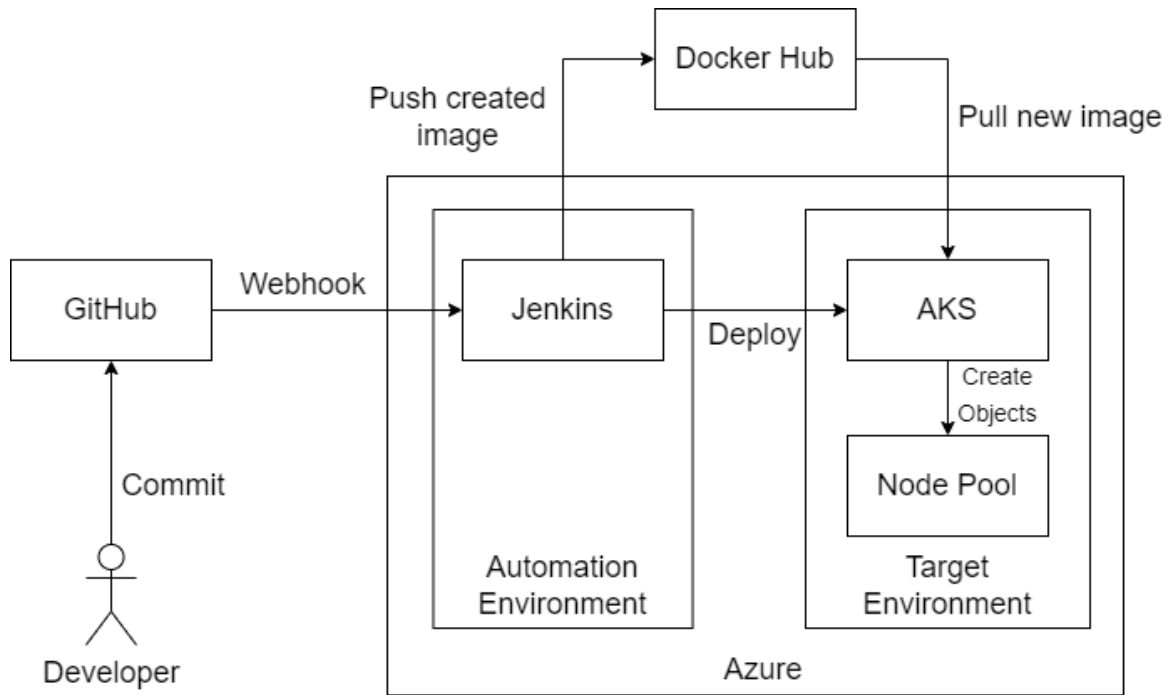
4. Project Implementation

In this chapter, performed will be all the necessary steps that will lead to the creation of the final system. It will start with setting up and configuring, the repository on GitHub and Docker Hub with the description of all necessary files. Then, cloud environments will be created. It will be followed by discussing Jenkins configuration and its pipelines. The chapter will finish by showing exact manual steps needed to complete the whole CI/CD and will be compared with the automated one, which also will be a demonstration, of how system performs.

Before diving into configuration processes, all selected technologies and platforms will be summed up. These are:

- VCS – Git, with GitHub,
- Container registry – Docker Hub,
- Application – Web application written in Java and Spring Boot,
- Cloud provider – Azure,
- Container Engine – Docker,
- Container Orchestrator – Azure Kubernetes Service,
- Build tool – Gradle,
- Automation Server – Jenkins.

To visually present the workflow, the scheme of the system is presented in Figure 4.1.

Figure 4.1 *Scheme of the implemented system*

4.1. Project Repositories

To start, original “Spring Petclinic” repository was forked. Inside that repository, three additional directories were created:

- `.circleci` – directory that holds configuration files for CircleCI, needed in the experiment from chapter 5.3,
- `jenkins` – inside this directory, four additional folders were created for every pipeline; each of the directories hold single Jenkinsfile; all Jenkinsfiles will be discussed in chapter 4.3
- `kubernetes` – inside this directory, there is file stored, called `manifest.yml` needed for creating Kubernetes objects; it will be discussed in detail later in this chapter.

The next thing that needs to be done is to create appropriate files and commit them to the repository, utilizing the idea of a single source of truth and pipeline as a code. The first file that needs to be created is `Dockerfile` in Figure 4.2. This file allows Docker to containerize the built application.

```

1 FROM adoptopenjdk/openjdk11:latest
2 EXPOSE 8080
3 ARG JAR_FILE=build/libs/spring-petclinic-2.6.0.jar
4 COPY ${JAR_FILE} app.jar

```

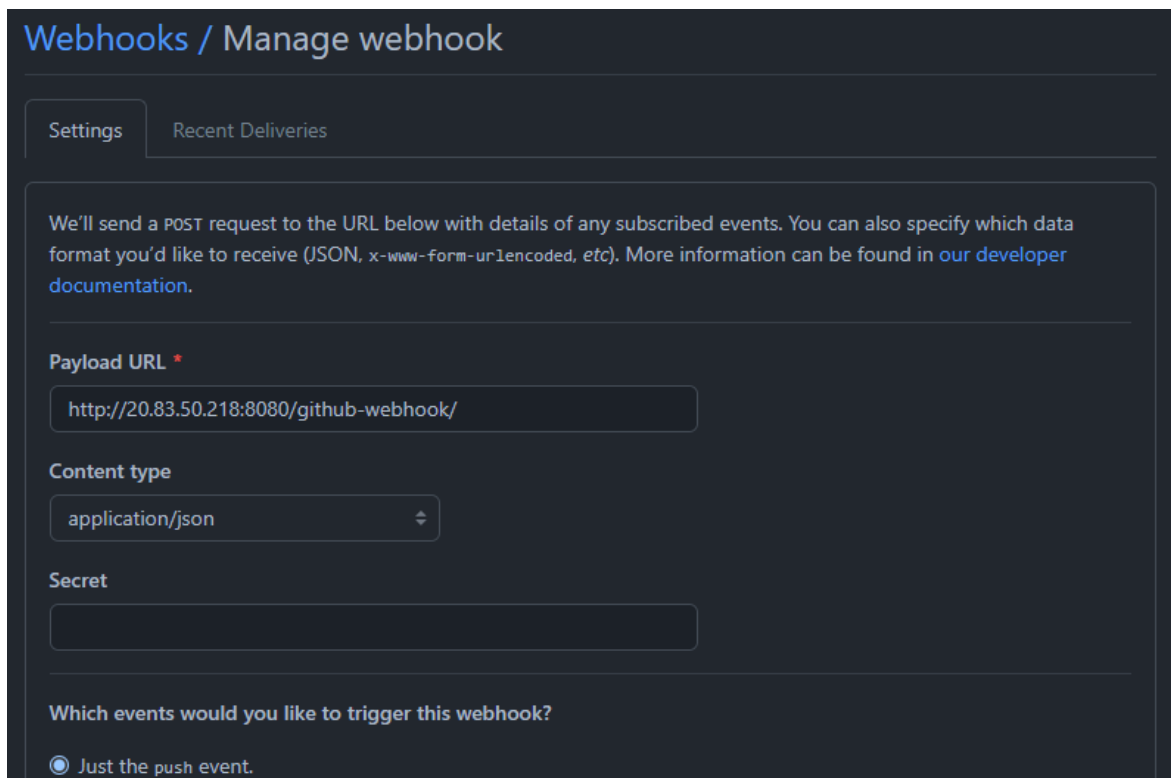
```
5 ENTRYPOINT ["java","-jar","/app.jar"]  
6
```

Figure 4.2 *Dockerfile for containerizing the application*

The first line tells Docker what base image to use. Then, it tells that port 8080 will be exposed. Line three specifies the variable for the artifact created during the build process. Line four copies mentioned file into the container. Last but not least, line five is defining the command that will start the application inside the container.

The next file, inside the “kubernetes” directory that needs to be created, is file called “manifest.yaml”. The naming of the file is up to the user. This file defines Kubernetes deployment, responsible for ensuring a correct number of running pods. It also contains the definition of the image with the latest tag. Another object it defines is a service that will expose application to the Public Internet.

When all necessary directories and files are created, the GitHub webhook must be set up to enable the possibility of triggering pipelines inside Jenkins automatically. From the repository level, the user needs to navigate to the “Settings” and then to “Webhooks”. Afterward, in payload URL, with the addition of “/github-webhook”, Jenkins URL must be pasted. An example configuration is shown in Figure 4.3.



The screenshot shows the 'Webhooks / Manage webhook' page in GitHub. It has two tabs: 'Settings' (active) and 'Recent Deliveries'. The main content area explains that a POST request will be sent to the specified URL with details of subscribed events. Below this, there are three fields: 'Payload URL' with the value 'http://20.83.50.218:8080/github-webhook/', 'Content type' set to 'application/json', and a 'Secret' field. At the bottom, there is a section titled 'Which events would you like to trigger this webhook?' with a radio button selected for 'Just the push event.'

Figure 4.3 *Webhook configuration from GitHub UI*

For better overview the repository is shown in Figure 4.4, excluding unnecessary directories and files. The image was generated with the usage of *tree* command where the “src” folder contains all application files.

```
.
├── master-thesis-spring-petclinic
│   ├── .circleci
│   │   └── config.yml
│   ├── Dockerfile
│   ├── LICENSE.txt
│   ├── build.gradle
│   ├── docker-compose.yml
│   ├── gradlew
│   ├── gradlew.bat
│   ├── jenkins
│   │   ├── build-and-deploy
│   │   │   └── Jenkinsfile
│   │   ├── delete-all-obj
│   │   │   └── Jenkinsfile
│   │   ├── init-build
│   │   │   └── Jenkinsfile
│   │   └── init-deployment
│   │       └── Jenkinsfile
│   ├── kubernetes
│   │   └── manifest.yaml
│   ├── mvnw
│   ├── mvnw.cmd
│   ├── pom.xml
│   ├── readme.md
│   ├── settings.gradle
│   └── src
```

Figure 4.4 *Output of the “tree” function*

The last thing needed to be created, in this chapter, is repository on the Docker Hub. It is important to note that, while pushing to the non-existent repository on Docker Hub it will be created automatically. Nonetheless, it is good practice to create it first in case of any additional configuration is needed.

Upon completion of all the tasks described in this chapter, further actions can be performed. They are concerned with all necessary deploying resources within chosen cloud platform and configuring them. All of this will be the topic in the next chapter.

4.2. Cloud Environments

Before anything will be deployed onto the Azure cloud, it is important to know its resource hierarchy and how they are created. Whenever the user wants to create any resources, he is communicating with a technology called Azure Resource Manager (ARM). ARM is Azure's deployment and management service. It gives a management layer that lets users add, modify, and remove resources within the user's account [50]. It doesn't matter if Azure portal, CLI, or other APIs will be used. All the methods invoke ARM behind the scenes. The scheme of this is presented in Figure 4.5.

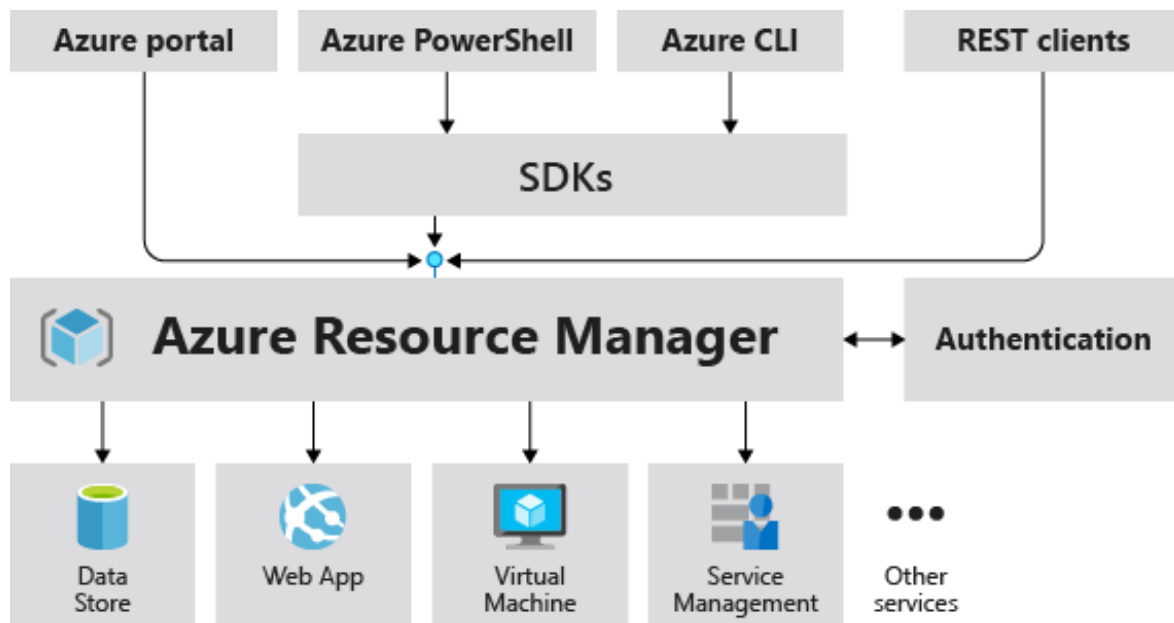


Figure 4.5 *Azure Resource Manager overview*,
Source: <https://learn.microsoft.com/en-us/azure/azure-resource-manager/management/overview>

Azure provides different levels of scope. These are management groups, subscriptions, resource groups, and resources, presented in Figure 4.6. Since management groups let organize and govern many subscriptions within organizations, they won't be used during this work. A subscription in Azure acts as a single billing unit for Azure resources. As a result, a subscription is an agreement between a company or single user and Microsoft to utilize resources, for which fees are paid either on a per-license basis or depending on the number of resources used in the cloud.

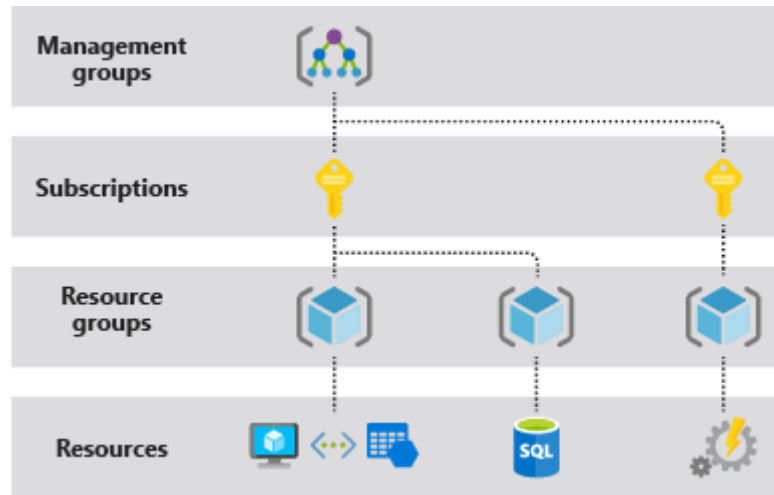


Figure 4.6 *Scope overview in Azure*,
 Source: <https://learn.microsoft.com/en-us/azure/azure-resource-manager/management/overview>

Resource Group (RG) is a container for an Azure solution that contains related resources. The resources the user wants to manage together are included in the same resource group. That means that every resource must be under RG and every RG must be under subscription. For example, in the discussed project there is a need for single a subscription. Then, two separate resource groups will be created – for automation environments and target environments, since they will be managed separately. Single resources will be deployed within each resource group.

First, a resource group with an appropriate name must be created. Its name, subscription it belongs to, and region must be specified. At this point, it is important to mention that the resource group's region doesn't have to be the same as the resource in it. Also, resources can be in different regions within a single resource group. Then, optionally, tags can be provided. After submitting all necessary information, Azure performs a review. This process checks if the given information is correct and if it does not violate any policies. After passing the review, RG can be created.

When the RG is created, a virtual machine can be deployed. Firstly, a user needs to provide basic information – a subscription, resource group, virtual machine name, region, OS image, and VM type. Next, a very important step needs to be performed, namely, specifying the Administrator account. Apart from the name, the most important is to choose between a Secure Shell (SSH) public key or password as an authentication method. If the SSH is chosen, Azure will create a new one with the given name that will be associated with the VM. If another key pair is already present in the resource group, it can be used for many different virtual machines. Since SSH is much safer than a password, this method will be chosen, with the default username which is azureuser.

Then, the disk's properties must be specified, but since no change is needed here, default options will be used. Networking is another step while creating the VM. Specified must be the virtual network the VM belongs to, a subnet, and public inbound ports. As it's the first VM within the resource group new virtual network must be created. Like SSH key pair, an already existing virtual network can be used. Port 22 will be publicly opened for communication with the VM since SSH operates over it. Also, for the communication new, a public IP address will be created. Leaving everything else as default, the review process has to be run once again. If it passes, the virtual machine can be created. Created resources for the automation environment are presented in Figure 4.7.




<input type="checkbox"/> Name ↑↓	Type ↑↓
<input type="checkbox"/>  jenkins	Virtual machine
<input type="checkbox"/>  jenkins-ip	Public IP address
<input type="checkbox"/>  jenkins-nsg	Network security group
<input type="checkbox"/>  jenkins164_z1	Network Interface
<input type="checkbox"/>  jenkins_key	SSH key
<input type="checkbox"/>  jenkins_OsDisk_1_c613eb33148a419baac5b1a2bc752a82	Disk
<input type="checkbox"/>  mklimas-automation-env-vnet	Virtual network

Figure 4.7 Azure resources for automation environment

Apart from the resources presented above, Azure automatically creates new resource group with a single resource in it, called Network Watcher. This technology helps to monitor, diagnose, provide metrics for resources in the virtual network [45]. For example, connectivity over a specified port can be checked between two different virtual machines. Nevertheless, to enable it, extensions must be installed on VMs.

Next, a target environment, with the managed Kubernetes cluster, must be created. The process is similar to the previous ones. The most important part of creating AKS is to configure Node Pool, which is the set of the same VMs, that will have installed all necessary Kubernetes components, and will be managed by the AKS control plane. As discussed, two virtual machines will be assigned to the Node Pool. The user can choose the preferred method of scaling: manual or autoscale. Manual will be used since autoscaling is not required. After the successful review process, apart from the Kubernetes Service inside the specified RG, Azure creates additional RG, with all necessary resources for the managed virtual machines. These resources were presented in Figure 4.8.








<input type="checkbox"/> Name ↑↓	Type ↑↓
<input type="checkbox"/>  ab3f43cc-bc00-49cd-8be0-3e47246f8c30	Public IP address
<input type="checkbox"/>  aks-agentpool-33793363-vmss	Virtual machine scale set
<input type="checkbox"/>  aks-agentpool-39696331-nsg	Network security group
<input type="checkbox"/>  aks-agentpool-39696331-routetable	Route table
<input type="checkbox"/>  aks-vnet-39696331	Virtual network
<input type="checkbox"/>  kubernetes	Load balancer
<input type="checkbox"/>  target_cluster-agentpool	Managed Identity

Figure 4.8 *Azure resources for managed cluster*

To sum up, four RGs were created, from which two of them were created automatically, by Azure. These are:

- NetworkWatcherRG – discussed earlier, this RG holds service for Network Watcher,
- automation-environment – environment with Jenkins automation server and all its components,
- target-environment – this environment holds only one resource which is Kubernetes Service,
- MC_target-environment_target-cluster_westeurope – this is automatically created RG by Azure; it contains all resources required for the Node Pool, where AKS will hold the application's load and is automatically connected with created Kubernetes service.

When all the resources are created, the next thing that needs to be done is to allow access to them from the Internet, because by default all incoming and outgoing traffic is blocked by Azure. For the Jenkins virtual machine, there's a need for adding two inbound rules that will allow inbound traffic on port 22, for connecting via SSH, and port 8080 to allow usage of Jenkins. Port for Jenkins can be configured to be any port, but since port 8080 is the default and there is no need to change it, port 8080 will be used. Also, these rules can allow traffic to all the public Internet or only specified users based on different variables, for example, public IP addresses.

Next, necessary applications must be installed on the Jenkins VM. To do this, the user needs to connect to the virtual machine. As SSH was chosen during the creation of the virtual machine, the SSH key needs to be downloaded and then the connection can be established with the command:

```
ssh -i <path-to-the-key> azureuser@<vm-public-ip-address>
```

After successful connection, installed programs are:

- Java Runtime Environment (JRE),
- Git,
- Jenkins,
- Docker,
- kubectl,
- Azure CLI.

With such prepared environments and resources, the next task is to configure the automation server. Apart from the configuration of Jenkins itself, appropriate pipelines must be created. All of those are the topic in the next chapter.

4.3. CI/CD Pipelines

Before starting to write Jenkinsfiles, which will contain all necessary steps for the deployment of the application, Jenkins also requires configuration. Once again, it will be necessary to login into a virtual machine, and then, the following steps will be performed:

1. Changing permission of the `docker.sock` to 666 to allow every user (or Jenkins in this case) to work with a docker. This can be achieved in different, more strict ways, but this simplification is sufficient in the case of this project.
2. Configuring the `kubeconfig` file, which is needed for communicating the cluster. This operation consists of a few additional steps:

- a. From virtual machine log in into Azure CLI with the usage of command:

```
az login
```

- b. After executing the command, the user is prompted to visit the given link outputted by the command and paste the given code. After that, the user can log in to his Microsoft account and can come back to the shell.

- c. Then the user needs to get the credentials (*kubeconfig* file), by invoking two below commands (they are prepared by Azure in the “Connect” tab from the AKS view):

```
az account set --subscription <subscription-id>
```

```
az aks get-credentials --resource-group <rg-name> --name <cluster-name>
```

- d. *Kubeconfig* file is saved in `/home/azureuser/.kube/config`, which needs to be copied to Jenkins into the appropriate directory. It is done with the command:

```
sudo cp ~/.kube/config /var/lib/jenkins/.kube/
```

3. The next step is to install the needed plugins. This requires login into Jenkins, available under `<VM-public-IP>:8080`, typed in the browser.
4. The last step is to add credentials to Jenkins for Docker Hub as a secret. It's needed because in order to push images Jenkins must log in to the Docker Hub, and since all Jenkinsfiles are stored under version control, credentials cannot be simply typed there.

After performing all necessary configuration, the simple pipeline was created to test if Jenkins can connect to the repository and if it's able to make changes to the cluster. The pipeline consists of two simple steps which are logging into the repository and then executing the command *kubectl get nodes* which, if Jenkins is able to connect to the cluster, should display a list of available nodes in the cluster. The job was executed and partial output was presented in Figure 4.9, confirming, that configuration was correct.

At this point, after all preparations, pipelines can be created and written. Each created pipeline, besides code in Jenkinsfile, requires configuring where pipeline definition is stored. As the GitHub is the single source of truth, each job within Jenkins was configured to use Jenkinsfiles from the GitHub repository, instead of writing them directly into the automation server. It requires changing the definition of the pipeline to “Pipeline script from SCM”. Then specified must be the repository URL, branch, and path to the Jenkinsfile in the repository.

The project consists of four pipelines. The first one is called *initial-deploy-to-cluster*. It is a very simple pipeline and consists of two stages, with single steps. The stages part of the pipeline is presented in Figure 4.10.

```

Login Succeeded
[Pipeline] {
[Pipeline] }
[Pipeline] // withDockerRegistry
[Pipeline] }
[Pipeline] // withEnv
[Pipeline] }
[Pipeline] // script
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Test kubectl)
[Pipeline] sh
+ kubectl get nodes
NAME                                STATUS    ROLES    AGE    VERSION
aks-agentpool-67922056-vmss000000  Ready    agent    7h41m  v1.23.8
aks-agentpool-67922056-vmss000001  Ready    agent    7h41m  v1.23.8
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] // withEnv
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
Finished: SUCCESS

```

Figure 4.9 Confirmation of correct Jenkins configuration

```

1  stages {
2      stage('Checkout') {
3          steps {
4              checkout([$class: 'GitSCM', branches: [[name:
5  '*/main']], extensions: [], userRemoteConfigs: [[url:
6  'https://github.com/Withel/master-thesis-spring-petclinic.git']]])
7          }
8      }
9
10     stage('Deploy to cluster'){
11         steps {
12             sh 'kubectl apply -f kubernetes/manifest.yaml'
13         }
14     }
15 }

```

Figure 4.10 Jenkins file for initial deployment

The first stage is checkout, which is downloading code from the GitHub repository. The second one called “Deploy to cluster” is applying an earlier written manifest for the Kubernetes. As in manifest image is always set to the latest version, it will download the most recent version of the image.

The next pipeline is called *remove-cluster-objects*. It has only one stage and step which is executing the command:

kubectl delete all --all

This deletes every object in the Kubernetes cluster. Such pipeline was very useful during the tests and demonstration purposes. Nevertheless, it’s often good practice to have a way of purging everything and then deploying “fresh” infrastructure in case of unrecoverable errors.

The third, and the most important pipeline is called *build-and-deploy*. Besides the same configuration as previous pipelines, it will be triggered by the commit made to a repository, thus, “GitHub hook trigger for GITScm polling” must be selected. It’s the main Jenkinsfile that builds the application and its image, pushes the image into Docker Hub, and then deploys a new image into the Kubernetes cluster. The pipeline begins with defining variables for the environment. These are shown in Figure 4.11.

```
1    environment {
2        registryCredentials = "DOCKER_HUB_CREDS"
3        dockerImage = ""
4        IMAGE_NAME="mklimas/petclinic:${BUILD_NUMBER}"
5        IMAGE_NAME_LATEST = "mklimas/petclinic:latest"
6    }
```

Figure 4.11 Variable for „build-and-deploy” pipeline

These variables, fulfill the following needs:

- *registryCredentials* – it defines credentials to the Docker Hub, that were set earlier, as a secret in Jenkins,
- *dockerImage* – it’s variable, that will hold reference to the created Docker image,
- *IMAGE_NAME* – it defines the full image name, along with the tag, which can be defined in different ways, for example, commit ID, timestamp, etc. For simplicity, the tag will be simply a build number, in this case.
- *IMAGE_NAME_LATEST* – it serves the same purpose as *IMAGE_NAME*, but instead of setting tag as a build number, it sets it to “latest”.

It consists of six different stages, which are presented in Figure 4.12. The first one is automatically added by Jenkins. This is the step where Jenkins downloads the definition of

Jenkinsfile from the repository. The second one is checkout, which has the same purpose as previously.

Stage View

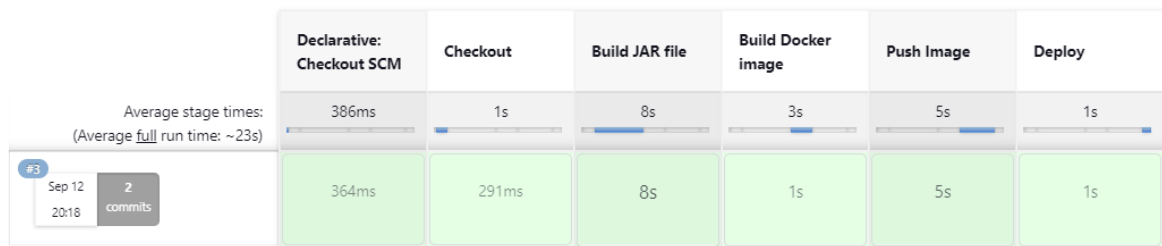


Figure 4.12 Stages in „build-and-deploy” pipeline

The third stage is called “Build JAR file” and is presented in Figure 4.13. First step of this stage executes the Gradle wrapper, which is stored in the repository. *Build*, in Gradle, as discussed earlier, builds the project and runs the tests. If the build is successful, results from the tests from the appropriate path are stored in Jenkins. It’s important to note that there is no need of archiving artifacts since they will be containerized in the next steps.

```

1      stage ('Build JAR file') {
2          steps {
3              sh './gradlew build'
4          }
5
6          post {
7              success {
8                  junit 'build/test-results/test/TEST-*.xml'
9              }
10         }
11     }

```

Figure 4.13 Stage „Build JAR file” in the “build-and-deploy” pipeline

The fourth stage, named “Build Docker image” and shown in Figure 4.14 build the image with the usage of the Dockerfile stored in the repository. It executes a simple script provided with the Docker plugin installed earlier. It assigns the name of the repository to the image and saves the image in the variable.

```

1      stage ('Build Docker image') {
2          steps {
3              script {
4                  dockerImage = docker.build("mklimas/petclinic")
5              }
6          }
7      }

```

Figure 4.14 Stage „Build Docker image” in the “build-and-deploy” pipeline

The next stage is called “Push Image” and is presented in Figure 4.15. With the help of the same plugin, as in the previous stage, first thing it does is log into the Docker Hub repository through the function `docker.withRegistry()`. The function takes defined credentials as an argument, and everything executed within it, will be done with specified credentials. Then, it pushes two of same images, but sets different tags for them. The first is tagged with the build number, and the second one indicates that this is the latest image. In the repository, the image with the “latest” tag will override the previous one. If pushing images is successful, it removes images from the Docker engine on a virtual machine.

```

1      stage('Push Image') {
2          steps{
3              script {
4                  docker.withRegistry('', registryCredentials ) {
5                      dockerImage.push("${env.BUILD_NUMBER}")
6                      dockerImage.push("latest")
7                  }
8              }
9          }
10
11         post {
12             success {
13                 sh "docker rmi $IMAGE_NAME"
14                 sh "docker rmi $IMAGE_NAME_LATEST"
15             }
16         }
17     }

```

Figure 4.15 Stage „Push Image” in the “build-and-deploy” pipeline

The last stage is called “Deploy”. Presented in Figure 4.16, the only thing it does is setting the image in the Kubernetes to the newest pushed image, based on the build number. Then, when the image definition is changed in the k8s cluster, the control plane downloads a newer image and starts a new container in a rolling fashion. Rolling fashion means, that old pods won’t be deleted until the new ones are started, ensuring maximum availability of the application.

```

1      stage ('Deploy') {
2          steps {
3              sh 'kubectl set image deployment/petclinic
4 petclinic=$IMAGE_NAME'
5          }
6      }

```

Figure 4.16 Stage „Deploy” in the “build-and-deploy” pipeline

The last pipeline is called *initial-build* and is almost the same as the previous one. It contains the same stages, except for “Deploy” stage. Also, during the build, it pushes image

only with the latest tag. Its purpose is to populate the repository, after its creation. It could've been achieved with the single pipeline, but for demonstration purposes and simplicity, it does its job perfectly.

With such a prepared repository, both environments, configured Jenkins, and written pipelines automated deployments can be performed. This, with the addition of actions needed to be performed during the manual deployment, will be described in the next chapter.

4.4. Deployments

First, all manual steps required to deploy a new version of the application will be listed. It is assumed that the system is already working, some images are already pushed, the repository is already pulled on the machine, and *kubectl* is also configured. This simplification is meant to simulate a situation when a new person joins a team and starts working on the development of the application. Also, based on these steps, the experiment in chapter 5.4 will be based. They are as follows:

1. The developer makes a commit and pushes it to the repository.
2. The developer builds the project with the command:

./gradlew build

3. The developer has to wait for the build to finish, then, to build the image, the developer executes the command:

docker build . --tag mklimas/petclinic:dev1

4. The developer has to wait once again, for the container to be built. When it's done, the developer needs to login into the Docker Hub, after the command is executed, he needs to open the web browser for authentication, unless credentials are already stored on the machine. The command responsible for logging is:

docker login

5. The developer pushes the image with the command:

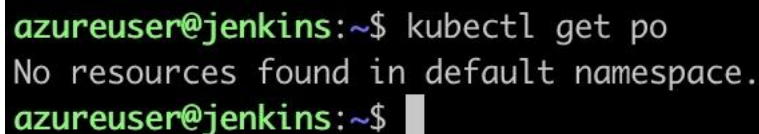
docker push mklimas/petclinic:dev1

6. The developer waits for the image to be uploaded onto the repository. When it's finished, the last command must be executed, that will switch the image in the Kubernetes cluster, which is:

kubectl set image deployment/petclinic petclinic=mklimas/petclinic:dev1

This whole process assumes that developer won't make any typos or mistakes, knows the appropriate naming of the tags, and does not perform any confirmation steps, like checking if the image was correctly built and tagged. It also can be noticed that in many steps the developer had to wait for the steps to finish.

Next, presented will be the same procedure with the usage of the described system. First, the container registry, along with the Kubernetes objects. After that, two initial jobs will be executed. They will prepare the necessary components for the environment. Before the execution of the jobs, output of the command *kubectl get po* is presented in Figure 4.17.



```
azureuser@jenkins:~$ kubectl get po
No resources found in default namespace.
azureuser@jenkins:~$
```

Figure 4.17 *Output of the command “kubectl get pods” before deployment*

After completing pipelines, two commands were run to see if the pods and services were created, which can be seen in Figure 4.18. Apart from that, in Figure 4.18, also IP address from the service, along with the port was copied and pasted into the browser to confirm if the application is publicly available. This also confirms that the image was successfully created and pushed to the registry.

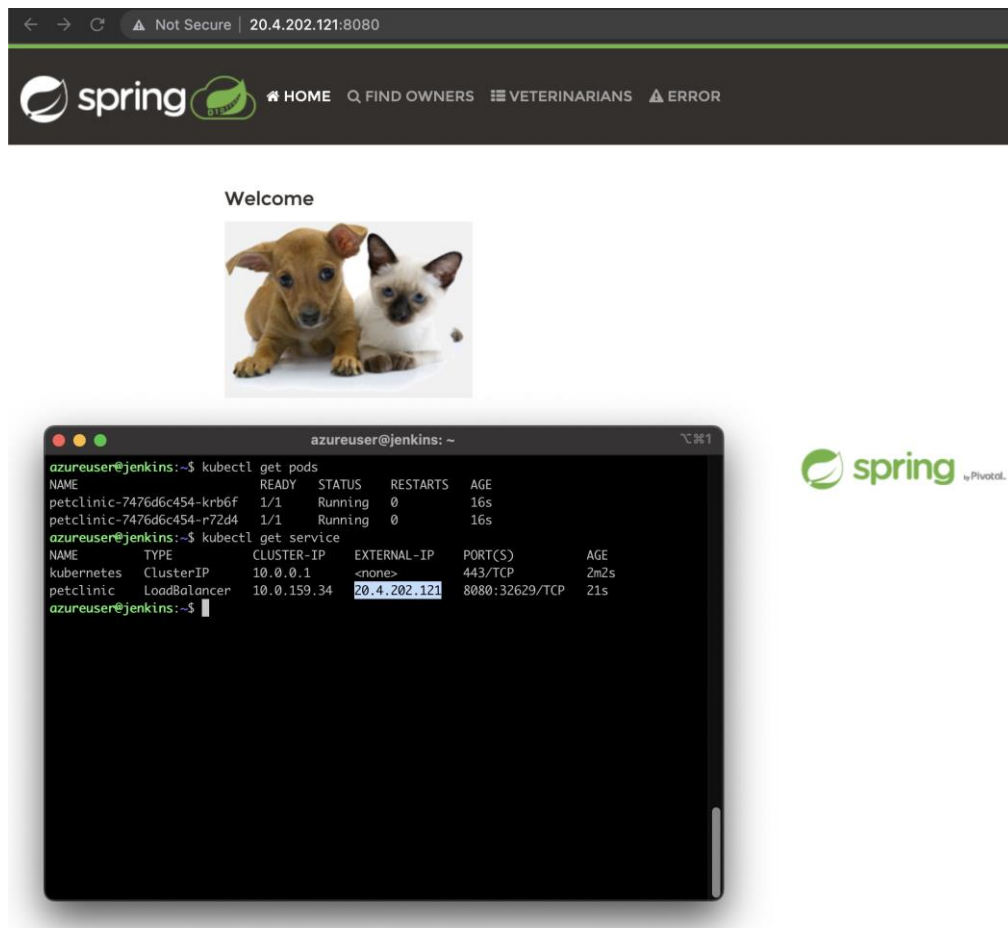


Figure 4.18 Output of command „*kubectl get pods*” after deployment

Next, the assumptions are the same as in the manual deployments, but the developer will only commit his work to the repository and the rest will be performed by automation. To demonstrate this default page, seen in the Figure 4.18 will be altered, by adding an additional picture of the rabbit. To do this, the developer must add the picture to the appropriate folder in the repository and write additional code responsible for displaying the image.

After pushing the changes, the *build-and-deploy* job is triggered automatically. It can be confirmed by examining the console output of the job, which says:

Started by GitHub push by <name>

Upon finishing the job and refreshing the “welcome” page in the application, a newly added image can be seen. This is presented in Figure 4.19.

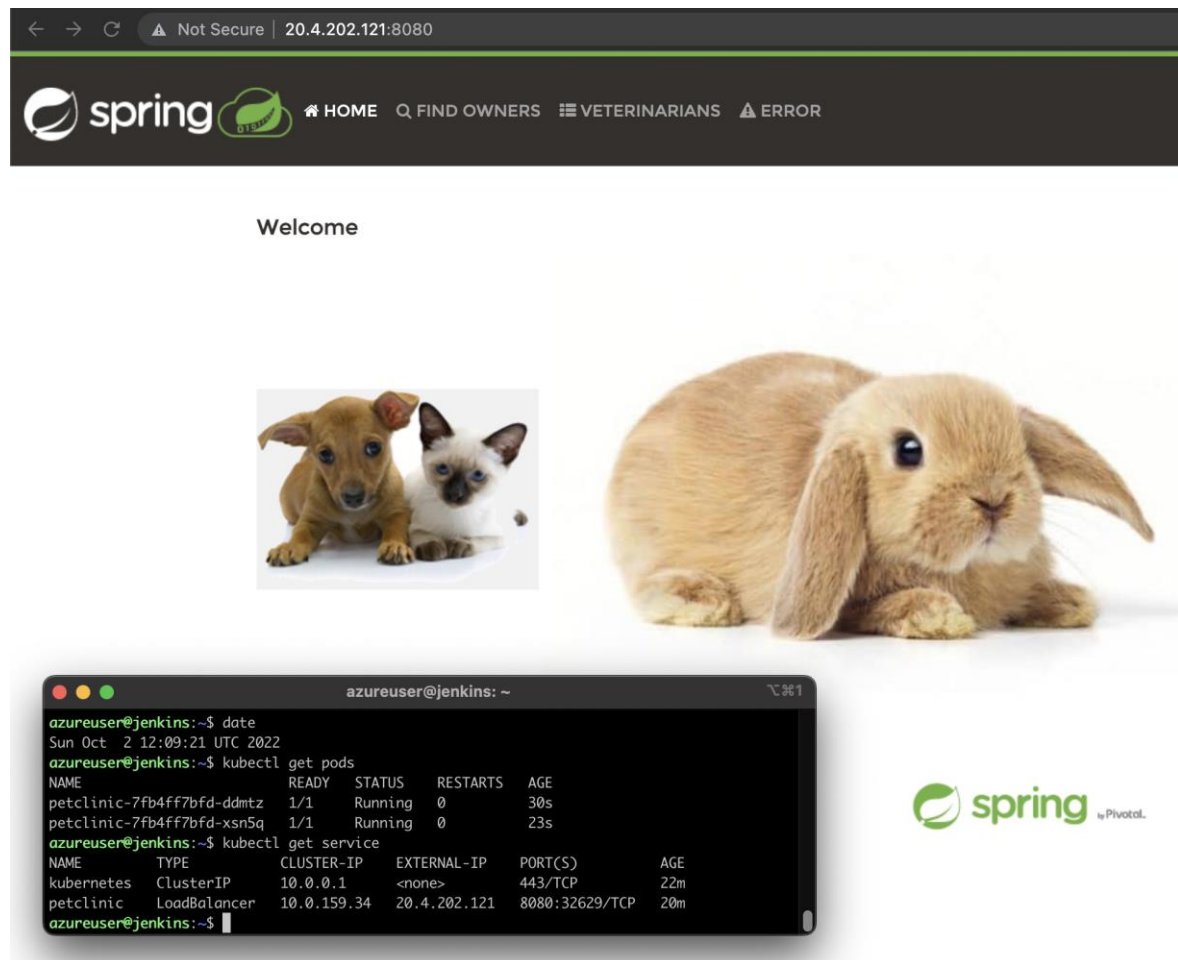


Figure 4.19 Change in the welcome page, after completing automated deployment

The whole process of deploying the new image to the cluster took approximately four minutes. The main difference is that during that time developers didn't have to wait for anything to finish. Also, they didn't have to know all the procedures behind the deployment process; they didn't have to check if everything was built and tagged correctly. Human input would be needed only in case the job failed.

This sums up the implementation and description of the project. This leads to the last topic, which is experiments which were performed during the implementation. They were important in the context of selecting appropriate technologies. All of them are described in the following chapter.

5. Experiments

To explain why some platforms or technologies were chosen over others, experiments had to be performed. In this chapter, all of them are described along with the commented results. The chapter starts with two tests that help determine which cloud provider is best suited for the needs of the project. Apart from that, the second experiment shows the importance of the choosing region where resources are deployed.

The third experiment compares the performance of CI/CD tools, based on which the choice was made. The last experiment shows the difference between time needed to perform manual deployments and the automated ones. Besides that, it helps highlight the importance of automation in software development projects.

5.1. Cost of the Resources

The first experiment refers to the prices of resources. The task of comparing resources plays an important role in lowering the costs of the project before any resources are deployed. It can also show the limitations of each cloud provider. Apart from the limitations, during the research, it also helps finding benefits of one platform over the other in case of available services.

Following all the considerations from chapter 3.2 estimations will be performed with earlier described, in the same chapter, calculators provided by Microsoft, Amazon and Google. Prices obtained from the calculators will be put into Table 5.1 and discussed afterward. A summary of all chosen resources with regions is shown below. Resources for Azure are:

- Three virtual machines 2 vCPU, 4 GB memory (B2s) in the West Europe region.
- Three SSD disks with 16 GB of storage in the West Europe region.
- One public IP address in the West Europe region.
- Managed Kubernetes Cluster (AKS) in the West Europe Region.

Resources for AWS for automation environment are:

- Three virtual machines 2vCPU, 4GB memory (t4g.medium) in the Europe Frankfurt region.
- Three SSD disks (gp2) with 16GB of storage in the Europe Frankfurt region.

- One IP address in the Europe Frankfurt region.
- Managed Kubernetes Cluster (EKS) in the Europe Frankfurt region.

Resources for GCP for automation environment are:

- Three virtual machines 2 vCPUs, 4GB (e2-medium) memory in the Warsaw region.
- Three SSD disks with 16GB storage in the Warsaw region.
- One IP address in Warsaw region.
- Managed Kubernetes Cluster (GKE) in the Warsaw region.

Estimates based on the above specifications are presented in Table 5.1 (prices are shown in USD per month).

Table 5.1 *Estimations of the required resources*

Resource / Provider	Azure	AWS	GCP
Virtual Machines	107.46	84.1	94.54
Disks	3.8	5.71	9.79
IP Addresses	2.63	~0	2.92
Kubernetes Cluster	0	73	0
Total	113.89	~162.81	107.25

Looking at the table GCP price estimations are the best. Nevertheless, some important aspects need to be highlighted. The first one is for the AWS IP address. Following the rule that the user pays only for the resources that he uses in the situation to reduce costs, for example, turning off (deallocating) virtual machines for the weekend when engineers won't be working on the servers, costs for the virtual machines will be reduced, but since IP address will be still reserved and not attached to the VM, AWS will charge costs for it. In other words, AWS is charging for the static IP address that is not attached to any resource.

While for virtual machines prices were similar, it's not the case with managed Kubernetes Cluster. The main difference is that Microsoft and Google offer free and paid versions of their clusters, which is not the case for the AWS. Of course, these tiers come with a downside, like decreased availability, worse performance, and others. Quoting Microsoft's documentation: "AKS recommends use of Uptime SLA in production workloads to ensure availability of control plane components. By contrast, clusters on the Free SKU tier support fewer replicas and limited resources for the control plane and are not suitable for production workloads. You can still create unlimited number of free clusters with a service level objective (SLO) of 99.5% and opt for the preferred SLO." [54] where uptime SLA is naming for the paid cluster. As the cluster for this project won't be holding any production workloads, the free tier is more than suitable to fulfill the needs.

Once again, it's important to highlight that these are only estimates and can differ from final charging and can differ even based on the time when they are made. Quoting Google's calculator page: "The estimated fees provided by Google Cloud Pricing Calculator are for discussion purposes only and are not binding on either you or Google. Your actual fees may be higher or lower than the estimate." [31].

This experiment shows the importance of such estimations. Without it, and necessary research it could've been easily missed that Azure and Google offer their managed k8s service in a free version. Realizing what is needed for the project in addition to knowledge about the offered services, in this case, using Azure or GCP allowed to reduce costs by about 30%.

5.2. Regions and Latency

The second test, regarding cloud providers, is meant to determine the best region for the project and show the difference between them. It shows the importance of deploying resources in the most suitable region, which can improve performance and reduce costs. Also, it is important to perform such research, because some of the resources might not be available in specific regions.

The following experiment will be performed: identical virtual machines (latency might also depend on CPU) will be deployed for each cloud provider in a region close to Poland and one far from it. The smallest available virtual machines will be used for the tests. These VMs have 1 vCPU and 1 GB memory (for GCP custom VM will be used).

The closest data center to the location where the project is being developed for respectively Azure, AWS, and GCP are West Europe Frankfurt (westeurope), Europe Frankfurt (eu-central-1), and Europe Warsaw (europe-central2). Relatively close to each other, but far enough, to demonstrate change in latency, from Poland for Azure, AWS and GCP respectively are Virginia United States (eastus), North Virginia, United States (us-east-1), and North Virginia, US (us-east4) [51],[52],[53].

To measure the latency commands described in chapter 3.2, which are *ping* and *curl* will be executed 100 times. For both commands, destination address will be the IP of the tested virtual machine, and the source will be a computer located in Poland, Silesia Region. After all preparations, experiments were performed, and the results of the experiments are presented in Table 5.2.

Table 5.2 *Regions latency comparison*

Provider & Region	Cost (USD/h)	RTT min	RTT avg	RTT max	TTFB min	TTFB avg	TTFB max
Azure West Europe	0.0074	36.501	44.301	242.369	0.0751	0.0934	1.1434
Azure East US, Virginia	0.0065	117.094	124.817	456.322	0.2318	0.2507	1.2560
AWS Europe Frankfurt	0.0134	36.400	46.253	70.697	0.075	0.0934	1.143
AWS US, North Virginia	0.0116	123.273	130.154	147.764	0.249	0.270	1.301
GCP Europe Warsaw	0.03	20.073	31.829	97.128	0.042	0.052	0.105
GCP US, North Virginia	0.03	121.923	129.252	148.436	0.245	0.282	0.254

After examining the table, it can be noticed that Google's results for the Warsaw region are the best in the case of latency. Azure and AWS Europe regions have almost identical results. The same situation happens in the case of data centers located in the United States. Nevertheless, differences between data centers located in Europe and data centers located in the US are much more significant.

This shows that the distance between the client and the server plays an important role in reducing the latency. If it is known that most of the end users of the application are located in a specific region, it is better to locate resources as close to them as possible to increase the end user experience.

Another conclusion can be made that the prices of the resources depend on the region in which they are deployed. Due to this, it makes it possible to make a decision based on which aspect is more important for the project. Resources can be deployed in regions where they are cheaper, sacrificing performance, or the other way around.

5.3. CI/CD Tools

The next experiment will help to choose the best combination of build tool and automation server for the project. The comparison of build tools and different automation servers will be performed as follows. Two Jenkins jobs and two CircleCI jobs will be created, one with the usage of Maven and the second one with Gradle. All the jobs will have to run the tests, build the project and when it's finished send it over SSH to another virtual machine. This will be performed five times with cached dependencies.

Comparison of build tools will be based on the performance of out of the box build process which for Maven is called *package* and for Gradle, it's *build*. No modifications to the environment, Java parameters, or build tool parameters were made. Change of the build tool will be the only difference in the pipelines run with the usage of Jenkins and CircleCI. The time needed for execution of each build will be captured with each builder output. Resource usage will be checked in virtual machine's metrics in Azure for Jenkins and for CircleCI metrics that are offered by the platform.

Evaluation of which automation server performs better is a bit more complicated. Since performance can depend on many different factors like the host's resources, used parallelism, or going deeper into caching configuration and these factors are far outside of the scope of this work. Thus, a comparison in this case will be made on basic, simplified performance and on the overall experience with using each application. Taken into account will be aspects like configuring a project, writing configuration files, or how convenient is to find information in the documentation.

First, both pipelines that use Maven and Gradle were run with the usage of Jenkins and CircleCI. Results obtained from the tests were put in Table 5.3.

Table 5.3 *Performance comparison of CI/CD tools*

Technology / Metric	Job Time	Build Time	CPU usage max	Memory usage max
Jenkins + Maven	25s	23s	48%	2GB
CircleCI + Maven	49.4s	26s	79%	2GB
Jenkins + Gradle	10s	7.3s	30%	800MB
CircleCI + Gradle	47.8s	21.4s	76%	800MB

Before analyzing the above results, it's important to point out that Jenkins runs as a single node application on a single VM with 2vCPU and 4GB available. CircleCI on the other hand ran all the jobs on "Large" class resources which have 4vCPU and 8GB of memory, which is two times more than Jenkins has. Even with such a disadvantage, Jenkins wins in almost all metrics.

Each metric will be analyzed separately since each one requires additional comment:

- Job Time – Compared is the total time that was required to finish the job. Jenkins outperforms CircleCI. It is happening mainly because CircleCI needs to create separate containers before running each job, while Jenkins runs them within its own process. On average the process responsible for creating new a environment took 8 seconds for CircleCI and restoring cache took on average 6 seconds. These numbers will be much less relevant while building much more complicated projects, but in the case of this work, Jenkins was much faster than CircleCI.

Nevertheless, in Timing tab for CircleCI, it suggests that timing can be improved by implementing parallelism.

- **Build Time** – this metric concerns specifically build tools. In the case of CircleCI, the difference between Gradle and Maven is almost not relevant, but with the addition of Jenkins, Gradle was more than three times faster than Maven. It is happening due to Gradle’s handling the same code in comparison to the last build (since changes were made mainly to configuration files). To confirm these additional experiments were run. Before running each build with the usage of Jenkins and Gradle change in the code was made. Then, on average build time took 22 seconds, which overlaps with the results from CircleCI. Still, Gradle was slightly faster than Maven.
- **CPU usage max** – Maximum metric was chosen because it shows how much CPU the process will need to run as fast as possible. It is important since in the case of running multiple processes, this will help avoid “Out of CPU” errors. In the case of Jenkins Azure’s metrics show that Gradle is more efficient than Maven in the context of this project. In the case of CircleCI jobs are running on the provider’s infrastructure, so the only important factor was that it didn’t reach 100%.
- **Memory usage max** – maximum metric was chosen for the same reason as in the case of CPU with the equivalent error “Out of Memory”. The same as previously for CircleCI, there was only a need to see if memory won’t reach 100% to avoid a lack of resources. In the case of Jenkins Azure metrics didn’t show any change during builds. It happened because Jenkins while running, occupies around 35% of memory, which is equal to 1.4GB, and the build were run within this capacity. The output of *top* command is present in Figure 5.1. Further experiments were performed with Jenkins service turned off and building project manually. In these tests, maximum value of memory used for Maven was 400MB and for Gradle 500MB.

```
top - 12:20:54 up 57 min, 1 user, load average: 0.00, 0.00, 0.00
Tasks: 111 total, 1 running, 110 sleeping, 0 stopped, 0 zombie
%Cpu(s): 0.0 us, 0.2 sy, 0.0 ni, 99.8 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
MiB Mem : 3925.3 total, 1482.1 free, 1534.0 used, 909.1 buff/cache
MiB Swap: 0.0 total, 0.0 free, 0.0 used, 2156.3 avail Mem
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
442	root	rt	0	280204	17796	8208	S	0.3	0.4	0:00.49	multipathd
671	jenkins	20	0	3718968	1.3g	31472	S	0.3	33.6	1:06.02	java
1	root	20	0	102656	11388	8208	S	0.0	0.3	0:01.58	systemd
2	root	20	0	0	0	0	S	0.0	0.0	0:00.00	kthreadd
3	root	0	-20	0	0	0	I	0.0	0.0	0:00.00	rcu_gp
4	root	0	-20	0	0	0	I	0.0	0.0	0:00.00	rcu_par_gp
5	root	0	-20	0	0	0	I	0.0	0.0	0:00.00	netns
7	root	0	-20	0	0	0	I	0.0	0.0	0:00.00	kworker/0:0H-events_highpri
10	root	0	-20	0	0	0	I	0.0	0.0	0:00.00	mm_percpu_wq
11	root	20	0	0	0	0	S	0.0	0.0	0:00.00	rcu_tasks_rude_
12	root	20	0	0	0	0	S	0.0	0.0	0:00.00	rcu_tasks_trace
13	root	20	0	0	0	0	S	0.0	0.0	0:00.05	ksoftirqd/0
14	root	20	0	0	0	0	I	0.0	0.0	0:00.36	rcu_sched
15	root	rt	0	0	0	0	S	0.0	0.0	0:00.01	migration/0
16	root	20	0	0	0	0	S	0.0	0.0	0:00.00	cpuhp/0

Figure 5.1 Output of the „top” command

5.4. Deployments

To highlight all benefits of the automation of the deployment processes, the following experiment was performed. Five people from the development and operations areas were asked to perform manual steps described in chapter 4.4 two times. None of them were familiar with the project, so the instruction on the steps was delivered to them. During that, they were requested to report the time they spent performing the steps, any encountered issues, and the naming they've chosen for the image tags. Results are presented in Table 5.4 and are discussed afterward.

Person, run	Time (min:sec)	Errors	Tag naming
Person 1, run 1	5:57	0	1
Person 1, run 2	2:36	0	2
Person 2, run 1	15:33	Docker Hub authentication	v1
Person 2, run 2	4:23	0	v2
Person 3, run 1	6:28	0	1
Person 3, run 2	3:11	0	2
Person 4, run 1	10:12	Communication issues	version-1
Person 4, run 2	3:45	0	version-2
Person 5, run 1	6:01	0	run-1
Person 5, run 2	3:21	0	run-2

Table 5.4 *Results of the manual deployments by people*

The first column shows the time each person needed to complete both deployments. The first thing that can be noticed is that the second run in every case was much shorter than the first one. It's mainly due to the fact that after the first deployment, all necessary dependencies and images were downloaded. Another aspect is that each person was using a different machine, which definitely can influence the time of the execution.

As shown, while presenting manual steps, the developer had to wait for some of them to finish, which is not the case in automated deployments. The average time in the following experiment was approximately 6 minutes and 11 seconds. Having five engineers working on the project, where each of them will make three commits every day, five days a week, will

result in 7 hours, 23minutes and 45 seconds of the idle time of the engineers. Of course, it is a big simplification, and there can be numerous variables that can change these times. Nevertheless, it shows that automated deployments allow saving the time.

Another benefit of automation comes from the fact that the developer does not have to know the processes behind the deployment. Apart from that, engineers also don't need to set up required platforms on their machines, as it was done for Jenkins virtual machine. In case of manual deployments instructions on how to perform such action must be written. On the purpose, these instructions didn't specify how to name the image, to show, that without automation, each engineer will need exact instructions on how to perform each step with the naming conventions.

As per errors, two out of five people had some issues, which required the help of a third party. The second error marked as "Communication issues" was purely a misunderstanding of the written instructions. This shows that training can be misunderstood by someone, resulting, once again, in wasted time.

The second error was during the execution of the command *docker login*. An issue occurred because the person already had cached other credentials in the Docker engine, which in the end, caused the troubles. For this experiment as an authentication method for Docker, the username and password were used. In real projects, it means, that each engineer will need to have created an account with the appropriate permissions. All of this can lead to security issues, while during the automated deployments Jenkins was, a bridge, between the user and the container registry along with the cluster.

This experiment was meant to show that implementing automation into deployment processes can eliminate a lot of different problems that can occur during the manual process. Nevertheless, creating such system is a complex and time-consuming task. The more complicated the deployment process, the more complicated the automation system, but also, the more complicated the deployment process, the more time can be saved by automating it.

6. Summary

This work aimed to show the impact of introducing automation into the software development lifecycle and the importance of the conscious selection of all the components of the automation system. All parts of the automation, as well as the target system were carefully analyzed, thus allowing to make the best possible choices for the project. After that, the system was implemented to compare it with the manual process. It also demonstrates the benefits that come from the implementation of such a system.

Even with all the simplifications made for this work, it was proven that automation plays an important role in a software development lifecycle. Also, this thesis shows how important is to know different solutions on the market, their limitations and advantages. With the appropriate research and experiments, meeting the project's needs becomes a much clearer task. It also helps to avoid unnecessary issues, caused by the lack of knowledge of the given system.

Further work in the area of automation could expand the work done in this thesis, by implementing more components used in a real, commercial project. These are for example web servers, separate database, or load balancers. Apart from that, automation of the creation of the environments can be performed. Similar to this work, different methodologies and tools can be analyzed in order to implement the best solution for the given problem.

Bibliography

- [1] G. Kim, P. Debois, J. Willis, and J. Humble. The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations. IT Revolution Press, Portland, OR, 2016
- [2] C. A. Cois, J. Yankel and A. Connell, "Modern DevOps: Optimizing software development through effective system interactions," 2014 IEEE International Professional Communication Conference (IPCC), 2014, pp. 1-7, doi: 10.1109/IPCC.2014.7020388.
- [3] K. Beck et al., "Manifesto for Agile Software Development", Agile Alliance. Retrieved, June 2001.
- [4] K. Gene, K. Behr and G. Spafford. The Phoenix Project : A Novel About It Devops and Helping Your Business Win. IT Revolution Press, Portland, OR, 2013.
- [5] S. Chacon and B. Straub. Version 2.1.356-2-g6e51aa7, 2022-09-21. Pro Git, Second Edition. Apress
- [6] S. Mysari and V. Bejgam, "Continuous Integration and Continuous Deployment Pipeline Automation Using Jenkins Ansible," 2020 International Conference on Emerging Trends in Information Technology and Engineering (ic-ETITE), 2020, pp. 1-4, doi: 10.1109/ic-ETITE47903.2020.239.
- [7] "What is CI/CD?", Red Hat,
<https://www.redhat.com/en/topics/devops/what-is-ci-cd> (access, 28.06.2022)
- [8] "Application Container Market - Growth, Trends, COVID-19 Impact, and Forecasts (2021 - 2026)", Research and Markets,
[https://www.researchandmarkets.com/reports/4845968/application-container-market-growth-trends?utm_source=GNOM&utm_medium=PressRelease&utm_code=qq42nv&utm_campaign=1502333+-+Global+Application+Container+Market+\(2021+to+2026\)+-+Growth%2c+Trends%2c+COVID-19+Impact%2c+and+Forecasts&utm_exec=jamu273prd](https://www.researchandmarkets.com/reports/4845968/application-container-market-growth-trends?utm_source=GNOM&utm_medium=PressRelease&utm_code=qq42nv&utm_campaign=1502333+-+Global+Application+Container+Market+(2021+to+2026)+-+Growth%2c+Trends%2c+COVID-19+Impact%2c+and+Forecasts&utm_exec=jamu273prd) (access, 30.06.2022)

- [9] R. Dua, A. R. Raja and D. Kakadia, "Virtualization vs Containerization to Support PaaS," 2014 IEEE International Conference on Cloud Engineering, 2014, pp. 610-614, doi: 10.1109/IC2E.2014.41.
- [10] A. M, A. Dinkar, S. C. Mouli, S. B and A. A. Deshpande, "Comparison of Containerization and Virtualization in Cloud Architectures," 2021 IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT), 2021, pp. 1-5, doi: 10.1109/CONECCT52877.2021.9622668.
- [11] Description of the basic advantages that come from using containers., Docker, <https://www.docker.com/resources/what-container/> (access, 03.07.2022)
- [12] I. M. A. Jawarneh et al., "Container Orchestration Engines: A Thorough Functional and Performance Comparison," ICC 2019 - 2019 IEEE International Conference on Communications (ICC), 2019, pp. 1-6, doi: 10.1109/ICC.2019.8762053.
- [13] A. Pereira Ferreira and R. Sinnott, "A Performance Evaluation of Containers Running on Managed Kubernetes Services," 2019 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), 2019, pp. 199-208, doi: 10.1109/CloudCom.2019.00038.
- [14] "Kubernetes – Architecture Overview", A. Patel, Medium <https://medium.com/devops-mojo/kubernetes-architecture-overview-introduction-to-k8s-architecture-and-understanding-k8s-cluster-components-90e11eb34ccd> (access, 11.07.2022)
- [15] "Kubernetes Components", Kubernetes, <https://kubernetes.io/docs/concepts/overview/components/> (access, 11.07.2022)
- [16] Home page of etcd project., etcd, <https://etcd.io/> (access 11.07.2022)
- [17] "Don't Panic: Kubernetes and Docker", J. Castro, D. Cooley, K. Cosgrove, J. Garrison, N. Kantrowitz, B. Killen, R. Lejano, D. "POP" Papandrea, J. Sica, D. "Dims" Srinivas, Kubernetes <https://kubernetes.io/blog/2020/12/02/dont-panic-kubernetes-and-docker/> (access, 12.07.2022)
- [18] "11 facts about real-world container use", datadog, <https://www.datadoghq.com/container-report-2020> (access, 21.07.2022)
- [19] "10 trends in real-world container use", datadog, <https://www.datadoghq.com/container-report> (access, 21.07.2022)

-
- [20] “IaaS vs. PaaS vs. SaaS”, Red Hat,
<https://www.redhat.com/en/topics/cloud-computing/iaas-vs-paas-vs-saas> (access, 24.07.2022)
- [21] “What is PaaS?”, Microsoft,
<https://azure.microsoft.com/en-us/overview/what-is-paas/> (access, 24.07.2022)
- [22] “What is SaaS?”, Microsoft,
<https://azure.microsoft.com/en-us/overview/what-is-saas/> (access, 24.07.2022)
- [23] “Serverless Computing”, Microsoft,
<https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-serverless-computing/> (access, 24.07.2022)
- [24] “What is private cloud?”, Microsoft,
<https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-a-private-cloud/> (access, 24.07.2022)
- [25] “As Quarterly Cloud Spending Jumps to Over \$50B, Microsoft Looms Larger in Amazon’s Rear Mirror”, RENO, NV, February 3, 2022,
<https://www.srgresearch.com/articles/as-quarterly-cloud-spending-jumps-to-over-50b-microsoft-looms-larger-in-amazons-rear-mirror> (access, 24.07.2022)
- [26] Repository that contains example project called “Spring PetClinic”, used for the purpose of this work,
<https://github.com/spring-projects/spring-petclinic> (access, 31.09.2022)
- [27] “Apache License, Version 2.0” Apache Software Foundation,
<https://www.apache.org/licenses/LICENSE-2.0> (access, 27.07.2022)
- [28] “Tomcat vs. Apache HTTP Server: What’s the difference?”, C. McKenzie, TechTarget,
<https://www.theserverside.com/video/Tomcat-vs-Apache-HTTP-Server-Whats-the-difference> (access, 06.08.2022)
- [29] Different types of Virtual Machines in Azure, Microsoft,
<https://azure.microsoft.com/en-in/pricing/details/virtual-machines/series/> (access, 10.08.2022)
- [30] Storage options for Compute Engine in GCP, Google,
<https://cloud.google.com/compute/docs/disks> (access, 11.08.2022)
- [31] Google Cloud Pricing Calculator, Google,
<https://cloud.google.com/products/calculator/> (access, 23.09.2022)
- [32] Description of free, or partially free, services for Azure, Microsoft,
<https://azure.microsoft.com/en-us/free/> (access, 11.08.2022)

- [33] Description of free, or partially free, services for AWS, Amazon,
<https://aws.amazon.com/free> (access, 11.08.2022)
- [34] Description of free, or partially free, services for GCP, Google,
<https://cloud.google.com/free> (access, 11.08.2022)
- [35] Regions and Zones in AWS, Amazon,
<https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-regions-availability-zones.html> (access, 15.08.2022)
- [36] “Region and availability Zones”, Microsoft,
<https://docs.microsoft.com/en-us/azure/availability-zones/az-overview> (access 15.08.2022)
- [37] “Using Apache Ant – Writing a Simple Buildfile”, Apache Ant,
<https://ant.apache.org/manual/using.html> (access 16.08.2022)
- [38] „Introduction to the POM”, Apache Maven Project,
<https://maven.apache.org/guides/introduction/introduction-to-the-pom.html> (access 16.08.2022)
- [39] “Introduction to the Build Lifecycle”, Apache Maven Project,
<https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html> (access, 16.08.2022)
- [40] List of the available plugins in Maven, Apache Maven Project,
<https://maven.apache.org/plugins/index.html> (access, 16.08.2022)
- [41] Basics of writing Gradle build scripts, Gradle,
https://docs.gradle.org/current/userguide/tutorial_using_tasks.html (access, 16.08.2022)
- [42] “Java in 2017 Survey Results”, E. Paraschiv, Baeldung,
<https://www.baeldung.com/java-in-2017> (access, 16.08.2022)
- [43] “The state of Java 2021”, JRebel,
<https://www.jrebel.com/resources/java-developer-productivity-report-2021> (access, 16.08.2022)
- [44] “Gradle vs Maven: Performance Comparison”, Gradle,
<https://gradle.org/gradle-vs-maven-performance/> (access, 16.08.2022)
- [45] Different distributions of Jenkins, Jenkins,
<https://www.jenkins.io/download/> (access, 20.08.2022)
- [46] Introduction to Jenkins Pipelines, Jenkins,
<https://www.jenkins.io/doc/book/pipeline/> (access 20.08.2022)

-
- [47] Pricing of the CircleCI, CircleCI,
<https://circleci.com/pricing/> (access, 25.08.2022)
- [48] Introduction to CircleCI, CircleCI
<https://circleci.com/docs/about-circleci> (access, 25.08.2022)
- [49] Introduction to Orbs in CircleCI, CircleCI
<https://circleci.com/docs/orb-intro> (access 25.08.2022)
- [50] “What is Azure Resource Manager?”, Microsoft,
<https://docs.microsoft.com/en-us/azure/azure-resource-manager/management/overview> (access, 03.09.2022)
- [51] “Azure Geographies”, Microsoft
<https://azure.microsoft.com/en-us/global-infrastructure/geographies/> (access, 03.08.2022)
- [52] Different Google Cloud Locations, Google,
<https://cloud.google.com/about/locations> (access, 03.08.2022)
- [53] “Regions and Availability Zones”, Amazon,
https://aws.amazon.com/about-aws/global-infrastructure/regions_az/ (access, 03.08.2022)
- [54] “Azure Kubernetes Service (AKS) Uptime SLA”, Microsoft,
<https://learn.microsoft.com/en-us/azure/aks/uptime-sla> (access, 06.08.2022)

List of abbreviations and symbols

<i>CI/CD</i>	Continuous Integration and Continuous Delivery
<i>SDLC</i>	Software Development Lifecycle
<i>VSC</i>	Version Control System
<i>CVCS</i>	Centralized Version Control System
<i>DVCS</i>	Distributed Version Control System
<i>CLI</i>	Command Line Tool
<i>GUI</i>	Graphical User Interface
<i>IT</i>	Information Technology
<i>VM</i>	Virtual Machine
<i>CPU</i>	Central Processing Unit
<i>OS</i>	Operating System
<i>OCI</i>	Open Container Initiative
<i>K8s</i>	Kubernetes
<i>CNCF</i>	Cloud Native Computing Foundation
<i>EKS</i>	Elastic Kubernetes Service
<i>AKS</i>	Azure Kubernetes Service
<i>GKE</i>	Google Kubernetes Engine
<i>CRI</i>	Container Runtime Interface
<i>DB</i>	Database
<i>IaaS</i>	Infrastructure as a Service
<i>PaaS</i>	Platform as a Service
<i>BI</i>	Business Intelligence
<i>SaaS</i>	Software as a Service
<i>IP</i>	Internet Protocol

<i>AWS</i>	Amazon Web Services
<i>GCP</i>	Google Cloud Platform
<i>CRM</i>	Customer Relation Management
<i>ERP</i>	Enterprise Resource Planning
<i>JVM</i>	Java Virtual Machine
<i>JSP</i>	JavaServer Pages
<i>API</i>	Application Programming Interface
<i>HTML</i>	HyperText Markup Language
<i>AI</i>	Artificial Intelligence
<i>IoT</i>	Internet of Things
<i>CDN</i>	Content Delivery Network
<i>NSG</i>	Network Security Group
<i>IOPS</i>	Input/Output Operations per Second
<i>B</i>	Byte
<i>vCPU</i>	Virtual Central Processing Unit
<i>I/O</i>	Input/Output
<i>NIC</i>	Network Interface Card
<i>RTT</i>	Round Trip Time
<i>TTFB</i>	Time To First Byte
<i>ICMP</i>	Internet Control Message Protocol
<i>XML</i>	Extensible Markup Language
<i>DSL</i>	Domain-specific language
<i>LTS</i>	Long-Term Support
<i>ARM</i>	Azure Resource Manager
<i>RG</i>	Resource Group
<i>SSH</i>	Secure Shell
<i>JRE</i>	Java Runtime Environment

List of Figures

Figure 2.1 DevOps methodology cycle, Source: https://www.dynatrace.com/news/blog/what-is-devops/	10
Figure 2.2 Scheme of VCS, Source: https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control	11
Figure 2.3 Scheme of CVCS, Source: https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control	12
Figure 2.4 Scheme of DVCS, Source: https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control	13
Figure 2.5 Stages of CI/CD, Source: https://www.redhat.com/en/topics/devops/what-is-ci-cd	16
Figure 2.6 Difference between virtualization and containerization, Source: https://www.redhat.com/en/topics/containers/containers-vs-vms	17
Figure 2.7 The relationship between Docker, CRI-O, containerd, and runC, Source: https://www.tutorialworks.com/difference-docker-containerd-runc-crio-oci/	19
Figure 2.8 Kubernetes Components, Source: https://kubernetes.io/docs/concepts/overview/components/	21
Figure 2.9 Change after deprecation of dockershim, Source: https://kubernetes.io/docs/tasks/administer-cluster/migrating-from-dockershim/check-if-dockershim-removal-affects-you/	22
Figure 2.10 Kubernetes share among container organizations, Source: https://www.datadoghq.com/container-report-2020/	23
Figure 2.11 Difference between deployment models, Source: https://www.redhat.com/en/topics/cloud-computing/iaas-vs-paas-vs-saas	25
Figure 2.12 Market Share of leading cloud infrastructure providers as of Q4 2021, Source: https://www.financialmirror.com/2022/02/11/amazon-aws-clinches-33-cloud-market-share/	27
Figure 2.13 Cloud Provider Market Share Trend, Source: https://www.srgresearch.com/articles/as-quarterly-cloud-spending-jumps-to-over-50b-microsoft-looms-larger-in-amazons-rear-mirror	27
Figure 3.1 Example view from Spring “PetClinic” application, Source: https://github.com/spring-projects/spring-petclinic	30
Figure 3.2 Three-tier architecture, Source: https://www.theserverside.com/video/Tomcat-vs-Apache-HTTP-Server-Whats-the-difference	31
Figure 3.3 Azure Cloud Console – Welcome page	34
Figure 3.4 AWS Cloud Console – Welcome page	34
Figure 3.5 GCP Cloud Console – Welcome page	35
Figure 3.6 Availability zones in Azure, Source: https://learn.microsoft.com/en-us/azure/availability-zones/az-overview	37
Figure 3.7 Example output of ping command	38
Figure 3.8 Curl command used in the experiment described in chapter 5.2	38
Figure 3.9 Example output of curl shown in Figure 3.8	39

Figure 3.10 Example Ant build.xml file	40
Figure 3.11 Example Maven pom.xml file	41
Figure 3.12 Example Gradle build.gradle file	42
Figure 3.13 Jenkins architecture, Source: https://www.jenkins.io/doc/developer/architecture/	45
Figure 3.14 Jenkins Home Page	46
Figure 3.15 Example of scripted pipeline in Jenkins	47
Figure 3.16 Example of declarative pipeline in Jenkins	48
Figure 3.17 CircleCI dashboard	49
Figure 3.18 CircleCI Architecture	50
Figure 3.19 Example pipeline in CircleCI	51
Figure 4.1 Scheme of the implemented system	55
Figure 4.2 Dockerfile for containerizing the application	56
Figure 4.3 Webhook configuration from GitHub UI	56
Figure 4.4 Output of the “tree” function	57
Figure 4.5 Azure Resource Manager overview, Source: https://learn.microsoft.com/en-us/azure/azure-resource-manager/management/overview	58
Figure 4.6 Scope overview in Azure, Source: https://learn.microsoft.com/en-us/azure/azure-resource-manager/management/overview	59
Figure 4.7 Azure resources for automation environment	60
Figure 4.8 Azure resources for managed cluster	61
Figure 4.9 Confirmation of correct Jenkins configuration	64
Figure 4.10 Jenkins file for initial deployment	64
Figure 4.11 Variable for „build-and-deploy” pipeline	65
Figure 4.12 Stages in „build-and-deploy” pipeline	66
Figure 4.13 Stage „Build JAR file” in the “build-and-deploy” pipeline	66
Figure 4.14 Stage „Build Docker image” in the “build-and-deploy” pipeline	66
Figure 4.15 Stage „Push Image” in the “build-and-deploy” pipeline	67
Figure 4.16 Stage „Deploy” in the “build-and-deploy” pipeline	67
Figure 4.17 Output of the command “kubectl get pods” before deployment	69
Figure 4.18 Output of command „kubectl get pods” after deployment	70
Figure 4.19 Change in the welcome page, after completing automated deployment	71
Figure 5.1 Output of the „top” command	77

List of Tables

Table 3.1 <i>Overview of the selected features of the automation servers</i>	44
Table 5.1 <i>Estimations of the required resources</i>	73
Table 5.2 <i>Regions latency comparison</i>	75
Table 5.3 <i>Performance comparison of CI/CD tools</i>	76
Table 5.4 <i>Results of the manual deployments by people</i>	78