

Master thesis



**Czech
Technical
University
in Prague**

Continuous Integration and Continuous Delivery of addictologic web application

Jana Čikelová

August 2021

I. Personal and study details

Student's name: **Čikelová Jana** Personal ID number: **452759**
Faculty / Institute: **Faculty of Electrical Engineering**
Department / Institute: **Department of Computer Science**
Study program: **Open Informatics**
Specialisation: **Data Science**

II. Master's thesis details

Master's thesis title in English:

Continuous Integration and Continuous Delivery of addictologic web application

Master's thesis title in Czech:

Continuous Integration a Continuous Delivery webové adiktologické aplikaci

Guidelines:

The topic of the work is a pre-existing web application that enables management of a addictologic programme, provides necessary APIs for iOS and Android apps presenting the programme and drives programme walkthrough. The application runs on Django python web framework with PostgreSQL as database and Huey/Redis to providing asynchronous multithread processing and scheduled tasks.

1. Study and analyze existing addictologic web application.
2. Prepare application for containerization using Docker environment and test fundamental functionality.
3. Study suitable container orchestration platforms for smaller projects with limited resources, e.g. Docker Swarm + Portainer, Kubernetes, Mickro8s. Use them to implement continuous integration and continuous delivery pipeline.
4. Explore state and performance monitoring tools to ensure high availability and provide means for tracking performance-related problems.
5. Choose suitable framework from appointed tools above and support your choice with arguments based on previous analysis.
6. Implement/configure the chosen solution.
7. Test the best framework on the real environment containing at least 5000 users

Bibliography / sources:

- [1]Erich, Floris & Amrit, Chintan & Daneva, Maya. (2017). A Qualitative Study of DevOps Usage in Practice. Journal of Software: Evolution and Process. 00. 10.1002/s
- [2]M. Shahin, M. Ali Babar and L. Zhu, "Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices," in IEEE Access, vol. 5, pp. 3909-3943, 2017
- [3]Arachchi, S A I B & Perera, Indika. (2018). Continuous Integration and Continuous Delivery Pipeline Automation for Agile Software Project Management. 10.1109/MERCon.2018.8421965
- [4]Sheyyab, Mahmoud. (2019). Managing Quality Assurance Challenges of DevOps through Analytics.
- [5] Khan, Muhammad & Jumani, Awais & Mahar, Farhan & Siddique, Waqas & Shaikh, Asad. (2020). Fast Delivery, Continuously Build, Testing and Deployment with DevOps Pipeline Techniques on Cloud. Indian Journal of Science and Technology. 13. 552-575. 10.17485/ijst/2020/v13i5/148983.

Name and workplace of master's thesis supervisor:

doc. Ing. Daniel Novák, Ph.D., Analysis and Interpretation of Biomedical Data, FEE

Name and workplace of second master's thesis supervisor or consultant:

Date of master's thesis assignment: **07.09.2020** Deadline for master's thesis submission: **13.08.2021**

Assignment valid until: **19.02.2022**

doc. Ing. Daniel Novák, Ph.D.
Supervisor's signature

Head of department's signature

prof. Mgr. Petr Páta, Ph.D.
Dean's signature

III. Assignment receipt

The student acknowledges that the master's thesis is an individual work. The student must produce her thesis without the assistance of others, with the exception of provided consultations. Within the master's thesis, the author must state the names of consultants and include a list of references.

Date of assignment receipt

Student's signature

Acknowledgements

First, I would like to express appreciation to my supervisor Daniel Novák and my colleges Jakub Trmal and Jindřich Prokop, which introduced me to the problematics of DevOps and guided me while working on this thesis. Furthermore, I would like to thank my family and friends for giving me a great environment and support during my studies.

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses

Prague, August 10, 2021

Abstract

Providing stable application with fast response to users' requirements is a key element in today's competitive world. Frequent deployment and high uptime are extremely valuable. In this diploma thesis, we focus on the upgrade of the server part infrastructure of smoke cessation web application to reach better stability and efficiency. Firstly we discuss the topic of containerization and virtual machines. Afterward, we dive into the concept of orchestration platforms, and we provide a wide comparison of the currently two most popular orchestration platforms Kubernetes and Docker Swarm. Additionally, we introduce UI interfaces for easier manipulation with these tools. We implement a Docker Swarm into our current infrastructure, and on top of it, we use Portainer UI. Furthermore, we discuss the topic of continuous integration and continuous delivery pipeline. Then, we design a fully automated GitLab CI/CD pipeline, so the deployment process can be performed without any human interaction. In the end, we present a solution for the performance monitoring of containers and the rest of the infrastructure.

Keywords: containerization, orchestration platforms, Kubernetes, Docker Swarm, CI/CD pipeline, container monitoring

Abstrakt

Stabilnosť aplikácie a rýchle reakcie na požiadavky užívateľov sú kľúčovými faktormi v dnešnom konkurenčnom svete. Časté nasadzovanie aplikácie a dobrý uptime sú mimoriadne cenené. V tejto diplomovej práci sa zameriavame na vylepšenie momentálnej architektúry serverovej časti aplikácie na odvykanie fajčenia s cieľom dosiahnuť lepšiu stabilitu a efektívnosť. Ako prvé prediskutujeme tému kontajnerizácie a virtuálnych strojov. Následne sa ponoríme do konceptu orchestrácie a poskytneme široké porovnanie momentálne dvoch najpopulárnejších nástrojov pre orchestráciu. Tými sú Kubernetes a Docker Swarm. Tiež predstavíme možnosti rôznych užívateľských rozhraní pre orchestráciu. Implementujeme Docker Swarm do našej infraštruktúry a na vrch neho pridáme užívateľské rozhranie Portainer. Ďalej preberieme tému continuous integration a continuous delivery. Potom navrhujeme riešenie plne automatizovanej GitLab CI/CD pipeline, vďaka ktorej sa proces nasadenia aplikácie bude diať bez zásahu človeka. Na záver predstavíme riešenie pre monitorovanie výkonu kontajnerov a zvyšku infraštruktúry.

Kľúčová slova: kontajnerizácia, orchestrácia, Kubernetes, Docker Swarm, CI/CD pipeline, monitorovanie kontajnerov

Contents

1 Introduction	1
2 Virtualization	3
2.1 VMs and Containerization	4
2.2 Docker	5
3 Orchestration platforms	9
3.1 Kubernetes	10
3.1.1 Lightweight Versions of Kubernetes	11
3.2 Swarm	12
3.3 Kubernetes and Swarm Comparison	13
3.4 UI managment	14
4 CI/CD Pipeline	17
4.1 Continous Integration	18
4.2 Continous Delivery	19
5 Performance Monitoring	21
5.1 Collection of Mertrics	22
5.2 Monitoring Tools	24
5.2.1 Sysdig Monitor and Promethues Comparison	24
6 Smoking Cessation Web Application	27
6.1 Logical Structure	27
6.2 Technology	29
7 Solution design and its implementation	31
7.1 Tuning of the upgrade and containerization	31
7.2 Cloud infrastructure management	32
7.2.1 AWS	32
7.2.2 Swarm Orchestration	33
7.3 GitLab CI/CD	34
7.4 Prometheus	36
8 Experiments and Results	39
8.1 Performance tests	39
8.2 Deployment improvement	42
9 Conclusion	45
9.1 Future work	46
A Bibliography	47

Figures

2.1 Comparison of hypervisor-based virtualization and containerization architecture [3]	5
2.2 Traditional Linux container versus Docker container [5]	6
3.1 Swarm service and its replicas [10]	13
4.1 The lifecycle of CI/CD pipeline [11]	18
5.1 Container monitoring architecture	22
5.2 Collection of metrics at different levels	23
6.1 Decision subtree of the application	28
6.2 The architecture of the app	29
7.1 Part of new admin interface. . . .	32
7.2 List of services deployed in the Swarm cluster	33
7.3 The diagram of the designed CI/CD pipeline	34
7.4 CPU usage of one of our containers is displayed in Prometheus.	37
8.1 The result of performance test of the old version of the application.	40
8.2 The result of performance test of the new version of the application.	40
8.3 Comparison of time response of an old and new version of the application.	41
8.4 The CPU usage of main computation unit	41
8.5 The CPU usage of database while read request were send to it	42
8.6 Comparison of the efficiency CI/CD pipeline and manual deployment	43

Tables

3.1 Summary of key differences between Kubernetes and Swarm . .	14
5.1 Summary of key differences between Prometheus and Sysdig Monitory	25



Chapter 1

Introduction

It is becoming more and more crucial to provide stable, efficient applications adaptive to the different amounts of traffic load. One of the approaches for reaching flexibility is to use containerization. Furthermore, the performance of the application is improved by setting it up to cloud infrastructure where it can run on multiple hosts simultaneously. To be able to handle these dynamic environments, orchestration platforms are used.


In addition, many mistakes done by developers are caused by repetitive manual tasks during deployment. Continuous delivery and continuous integration pipeline helps to minimize these mistakes. Once the process of deployment is automated, it allows developers to focus on the creative part of work, save human power and decrease needed financial resources. In this thesis, we will focus on techniques that help improve the stability and efficiency of the application. Furthermore, we will have a look at the automation of the deployment process.

In chapter 2, we will introduce the concept of virtualization. We will compare hyper-based virtualization and containerization and discuss their pros and cons. These two concepts are not mutually exclusive however, containerization is often viewed as the new, improved version of virtual machines. Furthermore, we will briefly talk about the currently most popular type of containerization - Docker containerization. In chapter 3, we will present orchestration platforms suitable for Docker containerization. Orchestration monitors the health of the containers, which can run on multiple nodes, and in case of problems, the traffic is redirected to the healthy node, and the problematic container is automatically restarted. Also, orchestration platforms help us to solve issues with load balancing and scaling. Nowadays, Kubernetes and Docker Swarm are the two most popular orchestration tools for Docker containers.

Chapter 4 will focus on the topic of the continuous integration and continuous delivery pipeline and its stages. By using the CI/CD pipeline, the deployment process is significantly simplified and less time-consuming. Because of automatization, fewer errors are introduced to deployment, and developers have more time to focus on other work. Furthermore, after the deployment of an application, it is still needed to monitor the health and performance of the application to make sure that everything runs smoothly.

Challenges of the performance monitoring when using containerization will be discussed in chapter 5. We will also introduce the currently best tools for performance monitoring on the market.

The practical part will be done on the server part of the application, which helps people to quit smoking through sessions with a virtual therapist. The logic of this application and its infrastructure is explained in chapter 6. Chapter 7 focuses on the solution design for this concrete application and its needs. We will show a detailed implementation of the Docker Swarm orchestration tool, GitLab CI/CD pipeline, and Prometheus container monitoring platform. In the final chapter, we will run tests to confirm improved efficiency. We will also compare the manual deployment with fully automated deployment.



Chapter 2

Virtualization

In the past, companies were hosting a single application per server, which was using only a fraction of the server's capacity. Also, the only possibility to improve efficiency and performance was through additional hardware. This approach led to higher operational costs. To save space and energy concept of virtualization was introduced. It increases flexibility, efficiency, scalability, and cost savings. When using virtualization, several applications on the same machine are simultaneously running in an abstracted layer from the physical hardware. Because these applications are in completely isolated environments, it occurs to them like they would be running on separate machines. Therefore it allows developers to test applications in various environments without the necessity of setting up several computers. The main advantages that virtualization brings to a company are:

- Operational flexibility
- Reduced cost
- Resiliency in disaster recoveries
- Higher productivity

There are two common ways to implement virtualization: Hypervisor-based virtualization and containerization [13]. They are not mutually exclusive giving the possibility to implement them separately or as a combination of both.

In the IT community, the term virtualization is commonly used in the meaning of hypervisor-based virtualization. Therefore, before we move to a further discussion, we need to define couple of terms for this thesis to avoid ambiguity.

Definition 2.1 (*Virtualization*). Virtualization is a process of running a virtual instance of a computer system in a layer abstracted from the physical hardware [2].

Definition 2.2 (*Hypervisor-based virtualization*). Type of virtualization providing virtual machines (VM), which requires the provisioning of an operating system [13].

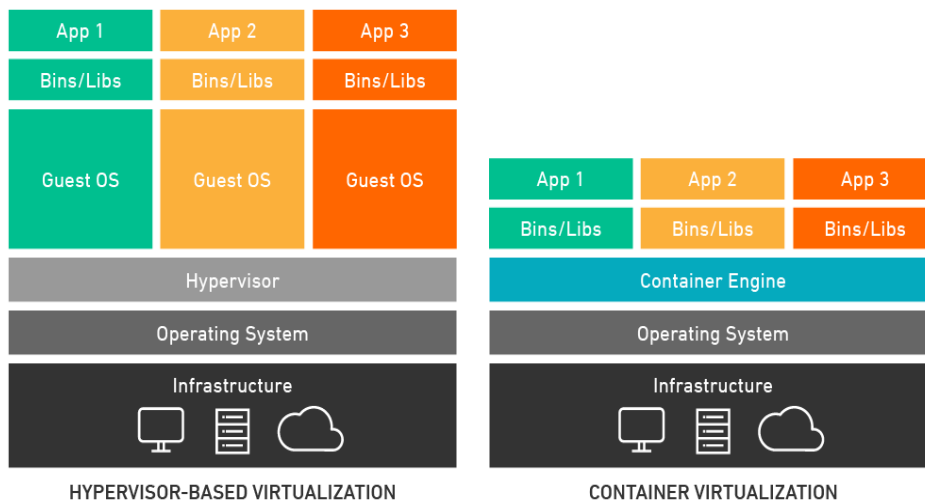


Figure 2.1: Comparison of hypervisor-based virtualization and containerization architecture [3]

side, VMs are a better choice when full functionality of OS is needed while running multiple applications on one server.

2.2 Docker

Docker is an open-source platform developed in 2013. It is the most popular tool for building, deploying and running Linux containers. Originally it was build based on LXC technology. But as it can be seen in fig. 2.2, compare to traditional Linux containers, in Docker containers, applications are broken into several processes. This division of applications brings several advantages beyond classical containers:

- Modularity
- Layers and image version control
- Rollbacks
- Fast deployment

Docker segmentation of the application into separate processes brings a possibility to take down only part of an application while working on an update or repair. Additionally, sharing processes between applications is achievable. Docker is very well suitable for microservice architecture however, microservices are not a requirement for it. Microservices divide “monolithic” applications into separate services. Microservices enable us to scale, manage, and updated services separately and asynchronously.

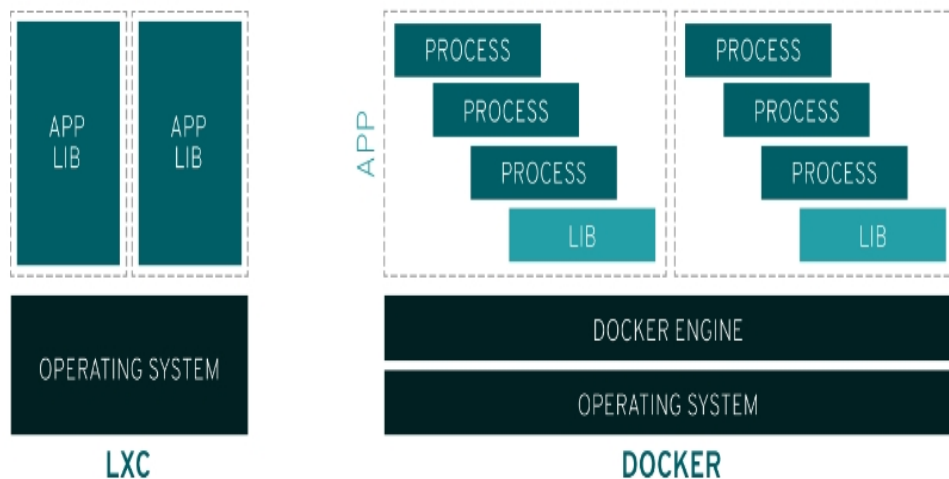


Figure 2.2: Traditional Linux container versus Docker container [5]

Docker containers are using a layering system. After the build of a container is initiated, a container engine goes through each line in the definition file and makes a layer, by adding the following instructions to the previous layer. This way a snapshot is created after every instruction. During the next build of the container, the cache is used for every instruction. If an instruction did not change, it is skipped by the container engine. The instruction that changed since the last build and its following instructions are applied. Additionally, the cache is used across containers, so if we have multiple containers, the order of instructions is very important, so the part of the instructions can be taken from the cache and used among various containers. So if we had another container on a machine where the image from 2.1 was already built into a container with the same working directory and requirements, the container engine is going to skip instructions until line 9. The cache mechanism allows to speed up the building process. Every time there is a change in the definition file, a new version of the image is created giving a user control over multiple versions of the image.

Image versioning makes quick rollbacks easier. If there is a problem with the current version, it is possible to rollback to the previous version of the application within a single command. This creates the perfect environment for agile development and continuous integration and continuous delivery(CI/CD) pipeline. The principle and advantages of CI/CD pipeline are further discussed in chapter 4.

Docker containers are famous for fast deployment. Containers give a possibility to share processes with the new app, which speeds up the building process. In past, provisioning and setting up hardware could take up to two days, nowadays with the Docker containers, we are able to deploy in seconds. There is no need to boot OS when moving or adding containers, therefore deployment time was significantly cut down. As a result the lifecycle of containers is much shorter and the deployment is happening more often which enables faster reactions to customer's feedback.

```
1 python:3.8-slim-buster
2
3 WORKDIR /app
4 #create working directory
5 COPY requirements.txt requirements.txt
6 #copy requirements into image
7 RUN pip3 install -r requirements.txt
8 # install dependencies
9 COPY . .
10 #add source code into image
11 EXPOSE 8000
12 #open port
13 CMD ["python3", "start.py", "runserver", "0.0.0.0:8000"]
14 #command to run inside of container
```

Listing 2.1: Example of Docker file

Docker alone has also some limitations. Once we start to have a bigger number of containers, it gets hard to manage and keep track of them. This and some other issues can be solved by orchestration platforms which we discuss in the following chapter. Also, using the Docker engine is a security issue. It requires root privileges, so it is important to be cautious about access data and the place where Docker runs.

Chapter 3

Orchestration platforms

Container orchestration manages the lifecycle of containers across multiple servers. It is especially essential in large, dynamic, micro-service platforms, in which containers are organized into clusters. When running an application in a cluster instead of on a single node, it is possible significantly to improve uptime, because when one node fails it is immediately replaced with a service from another node. Container orchestration gives us a possibility to deploy applications quicker, more often, and more efficiently, which is crucial because companies that are releasing their software often are having a higher chance to succeed in a highly competitive market [8]. Also, it allows us to scale our application dynamically. Apps using container orchestration tools can be managed by CI/CD pipeline, which brings the possibility of easier deployment over multiple environments. By using container orchestration tools, we are able to manage all containers in a cluster as one singleton. Cluster management has a complexity of $O(n^2)$ for placing m containers on n nodes [1]. The key abilities of container orchestration platforms are:

- Provisioning and management of clusters
- Scaling
- Load balancing
- Scheduling
- Health checks of containers
- Rolling updates and rollbacks

A wide range of orchestration tools is currently available on the market, however, they share some common features. The most popular choices nowadays are Kubernetes, Docker Swarm, and Apache Mesos. Typically there is a configuration file in which there is a description about where the orchestration tool can find images from which containers are built, networking between these containers, their mounted storage volume, and logs. The configuration file gives us as well the possibility to deploy several containers at the same time. Container orchestration platforms deploy containers and possibly their replicas to the respective hosts. When a new container is added

to the cluster, an event is scheduled for it. Certain conditions about CPU limits, memory availability, or metadata can be set. Once the containers are up and running, the container orchestration platform takes care of their lifecycle, and in case of failure of a container, it is started repeatedly.

```
1 version: '3.3'
2
3 services:
4 #list of services to deploy
5   db:
6     image: postgres:13
7     #image from which container is build
8     volumes:
9       - ../docker-data/postgresql:/var/lib/postgresql/data
10    networks:
11      - backend
12    deploy:
13      replicas: 1
14      restart_policy:
15        condition: on-failure
```

Listing 3.1: Example of configuration file in Swarm

Those tools, at which we will have a closer look and are popular within the Docker containerization platform are:

- Kubernetes
- Micro8s
- Swarm

Because of the current trend of high usage of orchestration tools and because the wide IT community is having lack of skills in setting up and administrating them, several cloud companies, such as Google, AWS, and Azure, are offering container orchestration services. These services are mostly Kubernetes based. Cloud-based orchestration offers various levels of management, starting from very basic as handling only network hardware, storage, and servers, up to full management when the customer interacts only through a web-based interface.

3.1 Kubernetes

Kubernetes is the most popular orchestration tool, which brings enormous support community. It is an open-source platform for efficient deployment, management, automation, and scaling of a containerized application. It focuses on modular orchestration, therefore it is suitable for any architecture deployment. Because of its robustness, it is a great solution for high-demand applications with complex configurations. On the other hand, the management

of the Kubernetes master requires specific knowledge, and for this reason, it is a too heavyweight solution for simple apps.

A Kubernetes cluster consists of master and worker nodes. Master controls all the resources in the cluster. Workers contain pods - small logical units which wrap one or more containers, which are managed by the controller. Containers within one pod share among themselves computational power, memory, IP address, port space, and specification about how to run the containers. Each pod has to have the following services: Docker, Kubelet, Kube-proxy.

On every node of a cluster, Docker is responsible for the preparation of images. This is accomplished either by the creation of images or by downloading them from the Docker hub, followed by starting of containers from these images. Kubelet is in charge of communication with the controller. It also guarantees that containers are up and running all the time. Kube-proxy serves as a network proxy, and so it provides communication with external users.

■ 3.1.1 Lightweight Versions of Kubernetes

Kubernetes is an excellent orchestration tool for large-scale deployments. But when it comes to smaller clusters or even a single host containerized apps, the robustness of Kubernetes becomes a bottleneck. When working with Kubernetes we should isolate from each other master and work nodes, a database for Kubernetes state should have its own instance, and nodes for incoming traffic should be separated. This approach makes sure that a large amount of load will be handle properly. However, many instances have to be started before starting our application. Therefore, following lightweight versions of Kubernetes were developed:

- K3s
- Minikube
- Micro8s

K3s is fully compliant lightweight Kubernetes, suitable for low-load applications. Backend storage based on the lightweight database SQLite. At the same time, it gives a possibility to connect to an external database endpoint. The supported databases are MySQL, PostgreSQL, and etcd - on which Kubernetes is based. K3s is so small in size that compared to Kubernetes it requires less than half of the memory [18]. Thanks to this fact, the installation process and deployment are faster.

Minikube is a single-node Kubernetes which runs inside a VM on a local machine. It supports almost all the features of Kubernetes. However, because it runs locally, cloud-based features such as load balancing or persistence of volumes are unavailable. Minikube needs only a relatively small amount of resources, and for this reason, it is a perfect fit for local development and experimentation.

Micro8s is a newer version of Kubernetes than Minikube, which was introduced in 2018 [12]. It is small, fast, multi-node, fully-conformant Kubernetes. The main difference from Minikube is that it runs locally on your machine with no virtual machine needed in between. Therefore it is appropriate for prototyping and testing. To reach an isolated environment from the local machine it packs all the binaries needed into a single snap package. Because a virtual machine is unnecessary, the consumption of resources such as RAM, CPU is much lower compared to Minikube. Compared to Kubernetes, Micro8s does not contain a controller in the cluster. All nodes connected to the Micro8s cluster are running as worker nodes. Once there are three or more nodes in the cluster, the high availability feature is automatically enabled, providing reliable service with zero downtime. Micro8s is a promising technology. However, it is still under development, not containing all the features. For example, availability on Windows or MacOS, was only added in 2020.

3.2 Swarm

Swarm is an orchestration tool designed by Docker, which is nowadays part of the Docker engine. Like Kubernetes, it can be used to manage, scale and deploy Docker containers. Any service which runs inside of the Docker container runs exactly the same in Docker Swarm mode. Through Swarm, we can easily manage, scale, and deploy a cluster of Docker nodes. In Swarm, various application environments can be efficiently maintained through multiple clusters. Swarm uses the same CLI (command line interface) as Docker, which makes it very simple to start Swarm once Docker is installed. Swarm may be an ideal solution for applications with a lower workload.

Similar to Kubernetes, Swarm consists of manager nodes and worker nodes. The communication between manager and worker nodes is direct. Swarm manager has two parts; scheduler and discovery service. The scheduler's job is to make sure that containers are running in their optimal states, which includes keeping the number of replicas up, accessibility to ports, and network. The manager node is also a worker node by default. However, it can be configured to not accept any workload. The discovery service determines if a node joined or left the cluster. Manager nodes assign one or several tasks to each of the worker nodes. Tasks are executed independently from each other, and in each of them, a single container is running.

Docker images are deployed through services. A service is generally an image within the context of a larger application. A service can run on one or more replicas of an image, as can be seen in fig. 3.1. A container image has to be defined before creating a service. Additionally, details such as the number of replicas, CPU and memory limits, rolling update policy, or behavior when service is restarted can be specified. Afterward, Swarm schedules tasks from service to nodes as one or more replica tasks. Service can be global or replicated. When service is global, it runs on every available node. For replicated service, manager schedules exactly the defined number of tasks and assigns them between available nodes.

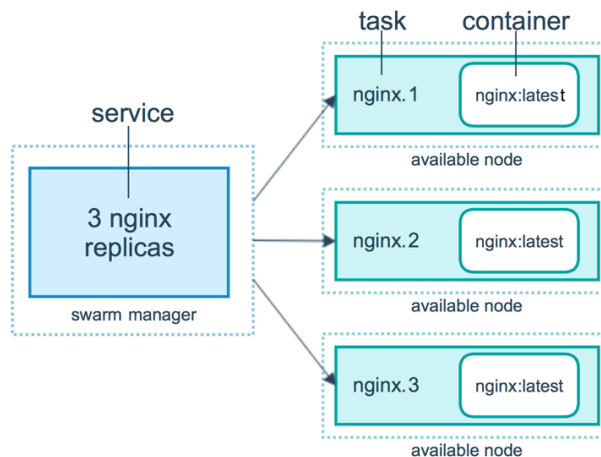


Figure 3.1: Swarm service and its replicas [10]

3.3 Kubernetes and Swarm Comparison

Both Kubernetes and Swarm can manage containers in a cluster, distributed workload between nodes, take care of frequent deployment and significantly improve uptime. As a result, it can appear that both of the platforms provide the same services and it can be hard to choose which of them is more suitable for a project. Therefore it is important to have a closer look on features such as installation, management, scaling, load balancing, storage in which they differ. Summary of the key differences can be seen in table 3.1.

Swarm is a part of the Docker machine. Once Docker is installed, it is possible to start Swarm default configuration with a single command through Docker CLI. In comparison to Swarm, starting a cluster in Kubernetes is quite challenging. Kubernetes has to be installed additionally to Docker. Also, network, ports, storage, IP addresses for pods need to be properly configured. On the other hand, when Kubernetes is managed by one of the cloud providers, no installation is required.

Because Swarm is directly integrated within Docker and uses the same CLI, it has a very easy learning curve. In comparison, Kubernetes has a large set of commands which makes it harder to start with. On the other hand, Kubernetes comes with its own GUI, which makes it easier to manage the clusters once the Kubernetes is running. Swarm does not have an integrated UI management system. However, there are available various UI tools which can be smoothly connected to Swarm. We will discuss this option in the 3.4.

One of the main reasons for orchestration is the possibility of handling higher loads of work. Therefore it is very useful to be able to scale up by adding VM. On the other side, when the demand is smaller, it is financially effective to scale down, so minimum resources are used. Since Swarm can deploy containers fast, it is possible to manually and quickly scale up or down the application. On the other side, Kubernetes supports horizontal auto-scaling based on servers traffic and automatically scales up or down as

needed.

It is not enough to have the correct amount of VMs, but it is also necessary to be able to distribute the work properly between them. In Swarm, all services are connected into one network, which allows connection from any node to any container. It has a DNS component that takes care of load balancing. Also, it assigns ports to services automatically or according to specific conditions of the user. In the case of Kubernetes manual configuration is required unless pods are exposed as services.

Swarm and Kubernetes have also various approaches to storage and sharing of volumes. Kubernetes allows distributing data volumes only between containers within one pod. In Swarm, it is possible to share data volumes between containers across all nodes. When a container is killed in Swarm also its data volumes are deleted. In Kubernetes, it is necessary to stop the whole pod to clear the data volumes of a container.

	Swarm	Kubernetes
Installation	Easy and fast	Manual configuration required
Managment	Docker CLI	Build-in GUI
Scalling	Scalling on demand	Horizontal auto-scaling
Load Balance	Internal load balancers	Manual configuration needed
Storage	Allows sharing between containers on any nodes	Allows sharing within a pod

Table 3.1: Summary of key differences between Kubernetes and Swarm

To conclude, Swarm is a part of the Docker engine with default configuration, which makes it easy to start up and use. On the other hand, Kubernetes is more robust. It is compatible with various types of containerization and therefore its popularity is higher. However starting with Kubernetes is more challenging, because it has its own language, and manual configuration at the beginning is required. Therefore Swarm is better for smaller projects, which need to manage fewer containers, and a high load of work is not expected. Kubernetes is more suitable for big projects, where specifying configurations is preferred. Also, a higher budget is essential since Kubernetes requires more manpower than Swarm.

3.4 UI management

Using Swarm alone and managing it only through CLI might be difficult and unnecessarily confusing, particularly when the cluster is more complex. This is a rising issue, especially because often, a wider range of developers on projects need to be able to access and manage containers without deep

insight into specific commands and configuration files. Therefore for easier management and better accessibility, various web-interface applications were developed. Many of them are also applicable to Kubernetes even though, by default, it has its own GUI. We will discuss the following most popular GUI for Swarm:

- Portainer
- Swarmpit
- Rancher

Portainer is a lightweight management UI tool for containerized applications [9] for maintaining different Docker environments. Portainer runs locally as a single container on a Docker engine which can run on Windows, MacOS, or Linux. It works with both Docker Swarm and all versions of Kubernetes. The UI of Portainer gives the user a possibility to build and publish images, deploy and manage applications. After deployment, Portainer gives you easy access to logs generated by containers, and in case the application fails, to fast redeployment. It is also possible to monitor basic performance such as CPU and memory. Portainer is a great tool if all you need is to manage a cluster. However, it is not very suitable for running more than one project simultaneously since the GUI is not well adjusted for it.

Swarmpit is an open-source lightweight GUI that can be added on top of Swarm. Similar to Portainer it runs as a single container on the Docker engine. It is currently available only for Linux OS. It provides management of services as well as stacks, networks, or volumes. Also, a user has access to real-time usage of resources such as CPU, memory, or disk. One of the best features of Swarmpit is the possibility to share the management console with the whole team since it allows multiple users for one Swarm cluster. Because Swarmpit is still a relatively new open-source project, some important features like autoscaling are not implemented yet.

Rancher offers complete management of clusters and acts as a frontend of a whole orchestration platform. It focuses on solving operational and security issues in a multi-cluster infrastructure, therefore it is great for medium to big environments with complex architecture. It runs on Linux OS and it is compatible with Swarm, Kubernetes as well as Cattle. Compare to previously discussed container GUIs, Rancher is far more complex with extra features such as changing the number of instances, host insights, or Docker machine drivers. Additionally to GUI, it also offers a CLI. Rancher is a robust tool and usually offers more than an average user needs. For this reason, a lightweight version of container GUI might be often more suitable to use.

Chapter 4

CI/CD Pipeline

In today's competitive world, the pressure to deliver software more often and at stable quality is increasing. However, rapid feature development and stability are in constant battle. On the one hand, introducing new features is bringing possible bugs into a steady system and on the other hand, trying to keep stability is preventing frequent software deployment. Also, humans are naturally failing at repetitive tasks and can not keep up their performance at a stable level. Therefore, new approaches how to automate the frequent and reliable deployment of software were introduced.

One of such approaches is continuous integration (CI) and continuous delivery (CD), also known as CI/CD pipeline. It brings automatization and monitoring to the software delivery process. The goal of CI/CD pipeline is to make it easier for developers to work simultaneously and deliver bug-free code several times per day. CI/CD pipeline builds code, runs tests, and deploys code to the production. It consists of series of steps that improve development as well as deployment of the application. Each of the steps can be executed manually, however, the goal of the CI/CD pipeline is to automate the process as much as possible. CI/CD pipeline makes the release cycle faster, reduces errors, and gives developers standardized feedback.

CI/CD pipeline consists of two main parts - CI and CD, and each of them contains several stages as can be seen in figure 4.1. The continuous delivery part can be further updated to continuous deployment [6]. Once this happens the whole process is fully automated including production deployment. The benefits gained by the implementation of CI/CD pipeline are:

- Automation of the software release process
- Higher quality of code
- Improved developer productivity
- Faster delivery of updates

As the code moves forward through the CI/CD pipeline, the quality of code is increasing. If for any reason one of the stages fails, the progression stops, and the results are sent back to developers. Therefore to be able to start any stage, all of the previous stages have to end successfully. As a result of this

process, it is guaranteed that only code which passed all tests is deployed to production.

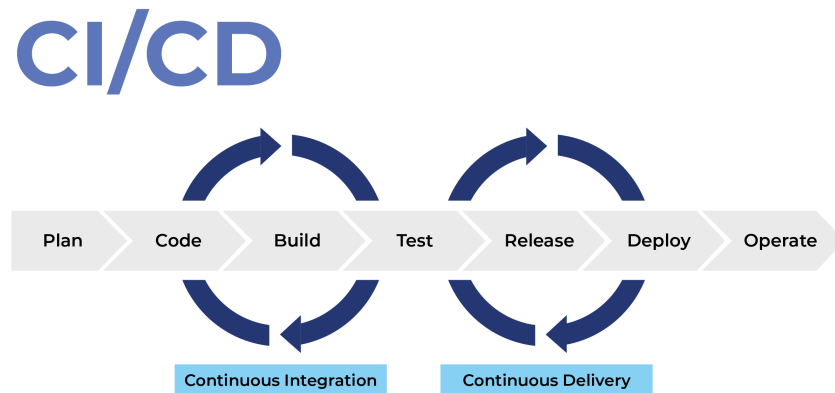


Figure 4.1: The lifecycle of CI/CD pipeline [11]

A script can be build to initiate an automatic process of whole or part of CI/CD for every commit, even in development branches. This includes stages such as building, testing, and deploying. Each push to the repository can have a tag and therefore multiple scripts can be built for various situations. By using these scripts, the chance of bringing an error to the application is decreasing.

4.1 Continuous Integration

The first part of the automated pipeline is continuous integration. Developers using the CI technique commit their changes to the common repository on regular basis, usually at least once per day. This allows several developers to work on the app simultaneously while minimalizing the chance of conflicts between commits of different developers. After changes are integrated into the central code base, automated builds and multiple tests are executed. Bugs are detected and easily fixable. Because hundred of tests are performed by CI server automatically, tests costs are reduced massively. After the code runs through the whole CI, developers receive feedback. This process should not run more than 10 minutes. Continuous integration generally consists of the following stages:

- Code
- Build
- Test

During the code stage, several developers are coding and committing simultaneously to a shared repository. Before starting a full build, the code gets first checked for static policies, during which syntax errors are revealed.

This cuts down resource utilization and time. In the build stage, the automatic build of committed code in the staging environment is executed. The staging environment is a replica of a production environment. It makes sure that newly committed code is compatible with the rest of the application and its libraries and that the build will always work. The executable files and SQL scripts are created and tested along with other configurations files.

During the test stage, automated tests are performed to check the functionality and performance of the app, which reduces the cost of testing significantly. Based on the size of the build, this process can take anywhere from seconds to hours. To save time, large corporations are running these checks in parallel environments. Integration tests investigate if parts of an application, and also an application as a whole, fulfill specified functional requirements. Load and stress tests are checking the stability and performance of an application during high load traffic. These tests are not usually run after every small update because full-performance tests can take a significant amount of time. Instead, they are executed when a major release is coming or after a bigger group of updates was committed.

4.2 Continuous Delivery

The next phase after automated builds and tests is continuous delivery. It makes sure that the validated code is released to selected environments of application as soon as possible. This process can be fully automated or with manual steps at critical points, such as triggering deployment. When the deployment does not need any more human intervention, we talk about continuous deployment. Typically, the teams are working with more than just a production environment. Usually, the changes are first deployed into the testing and developing environment. Continuous delivery makes sure that new features can be delivered faster and more often based on customer feedback. The main phases of CD are [7]:

- Binary Packet Manager
- Staging
- Production

In the binary packet manager stage, binary packages of required dependencies are combined with already successfully build code. Also, a configuration file with database changes and other infrastructure updates is added.

Next, the code is deployed to the testing/stage environment, where further tests are performed such as testing against the corresponding database, production acceptance test, etc.

After passing all the tests the code is ready to be deployed to the production environment. Not every commit gets deployed to production right after passing all the tests in the stage environment. Sometimes is preferable to group several updates before releasing a new version of the application.

Chapter 5

Performance Monitoring

Projects based on containerization can consist of anywhere between a few up to several hundreds of containers. We need to understand the health of whole applications but at the same time being able to break down the statistics into singular containers to detect issues fast. Container monitoring is a critical step for the optimization of performance. To ensure peak app performance, real-time monitoring is needed. The main advantages of container monitoring are:

- Early detection and solution of issues
- Safe implementation of changes while keeping an eye on the entire environment
- Improved performance and better user experience
- Resource allocation optimization

The possible complexity of container structure and dynamic allocation of resources brings challenges to performance and log monitoring. Therefore it is crucial to be able to deploy performance monitoring tools quickly across all the nodes. Since containers create a dynamic environment, container monitoring tools are sending collected data into a centralized location before they vanish when a container dies. Containers make monitoring problematic because they act like isolated "black boxes". The main challenges of container monitoring are:

- Addition of the second service to a container - The key idea of containers is having only one service per one container. Adding a monitoring process inside of a container to obtain inner visibility is in direct conflict with the main advantage of container - simplicity.
- Scalability of containers - Containers can move around and scale up or down. Therefore it is not possible to point monitoring tool to relevant applications. Additionally, we want to monitor the performance of the whole service, which can include several containers, as well as the metric statistics of single containers.

- Long-term analysis - The host, network, container, and orchestration platforms provide information regarding the performance and health of the application. Because the data are coming from different sources, the cardinality of metrics grows exponentially [22]. Therefore for long-term analysis and trends, horizontal scaling is required.

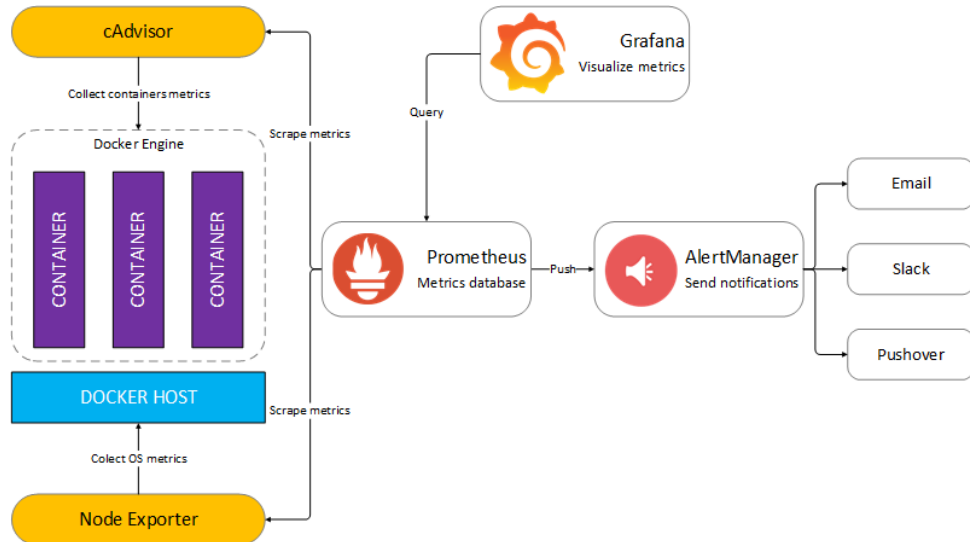


Figure 5.1: Example of a solution of container monitoring including graphical visualization and alert manager [14]

Usually, performance monitoring does not end with collecting various types of metrics. For a proper usage and understatement of statistics by developers, we need to visualize them. This is usually done by an additional platform as it can be seen in fig. 5.1. As a final step alert manager should be added, which sends notifications to various communication canals. This way developers can react quickly to arisen problems.

5.1 Collection of Metrics

To obtain a real-time response, resource utilization, and overall health of the application, we need to combine several types of metrics and event logs. It is important to choose the correct technique or combination of techniques suitable for a concrete project. The available approaches are:

- Monitoring process inside of a container
- Sidecar containers
- Agent-per-pod
- Syscalls

When monitoring a process inside of a container, a manager agent service is to a container, which collects and export metrics. However, this is quite an inconvenient solution because containers were designed to include preferable only one service.

The sidecar container approach appends a separate monitor container to an existing container of application. Therefore, a monitor agent runs in its own isolated environment. However, because to every container a monitor container is attached, the overall amount of containers doubles.

When using the agent-per-pod technique, a monitoring agent is attached to a group of containers, in the case of Kubernetes to a pod, which shares the same namespace. This approach is relatively easy to set up. However, the resource consumption of a monitoring container agent is quite big because of the high flow of data through the agent.

Syscalls approach assigns one agent per host. The agent collects all system calls which are traveling through the OS kernel. Therefore, it is possible to see the inside of a container from the outside. Syscalls method reduces agent's resource utilization and does not complicate container infrastructure. It collects metrics of single containers, short-lived processes, orchestration tools as well as overall infrastructure.

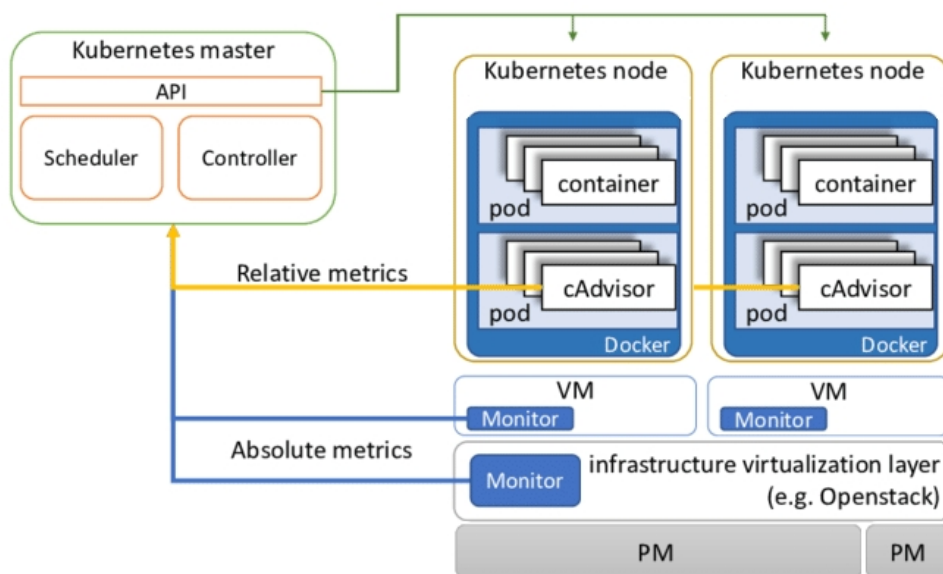


Figure 5.2: Diagram of collection of metrics at different levels of Kubernetes infrastructure [15]

Containers solutions collect data according to logical rather than physical infrastructure. Therefore, the metrics can be collected at different levels of the infrastructure hierarchy. As it can be seen in fig. 5.2, in Kubernetes we get information about specific containers, pods, services, host, etc. In case of a problem, we can drill down from the specific pod to process to identify where exactly the problem is occurring.

■ 5.2 Monitoring Tools

There are several aspects that need to be considered when choosing a monitoring tool. Configuration and deployment difficulty, detail level of metrics, visualization and its dashboard, alert configuration, and cost of the tool, all of these questions need to be answered. Even though there is a broad variety of tools on the market, Sysdig Monitor and Prometheus are the two most widespread solutions. Both of them are very well compatible with Kubernetes and Docker.

Sysdig was originally a troubleshooting tool used for Linux hosts as well as for Kubernetes and Docker. It evolved into two branches Sysdig Monitor and Sysdig Secure. Sysdig Monitor is a commercial tool for monitoring, alerting, and troubleshooting containerized environments specifically focused on Docker and Kubernetes. Sysdig Monitor's backend is very scalable, and therefore it is a great solution with a large number of nodes.

Prometheus is open-source monitoring and alerting tool. It uses a time-series database. Because of its database, it is a great tool for modern distributed systems and it is not limited only to container monitoring. However, additional software like Grafana and AlertManager are required to build a complete monitoring infrastructure.

■ 5.2.1 Sysdig Monitor and Prometheus Comparison

We will compare the two most popular solutions, which are Sysdig Monitor and Prometheus, from a technical point of view. We will have a closer look at instrumentation, data types, query language, alerting, and management.

Prometheus instrumentation is based on the agent-per-pod approach. But the resource consumption can be quite high if there are too many pods in the infrastructure. Also, every pod has to be configured to be possible to add a monitoring agent. However, Prometheus is still relatively easy to set up. In comparison, Sysdig Monitor uses the syscalls method, which means that it installs one monitor agent per each host. A transparent instrumentation layer is added between containers and the host. This layer collects all kinds of metrics and sends them to the monitor agent for further processing and forwarding. Both Prometheus and Sysdig Monitor also offer a possibility of direct instrumentation of code, while providing several libraries for it.

When comparing data types and formats, Prometheus and Sysdig are equally supporting Prometheus format and metadata. Sysdig Monitor additionally provides JMX and StastD data. In Prometheus, these kinds of data can be obtained with additional configuration. Sysdig Monitor also collects events as well as incident response records which is a remnant of Sysdig Monitor being originally a troubleshooting tool.

Prometheus offers a functional expression language through which we can select, aggregate, and manipulate collected data. The result of a query is either visualize in Prometheus's browser or send to an external system. On the other hand, Sysdig Monitor does not offer any kind of query language.

Alerting is an important part of processing metrics in container monitoring. Prometheus is equipped with a component called `ALertManager`. Alerts are built directly from Prometheus queries, which gives us an opportunity for pretesting. In Sysdig Monitor, alerts are managed through the GUI interface. Because it collects information not only about metrics but also events, Sysdig Monitor's main advantage is that it alerts about both events and metrics.

When it comes to the management of data Prometheus and Sysdig Monitor are having completely different approaches. Sysdig Monitor management is based on the idea that developer and operational teams need different access to data. Therefore it gives a user an individual dashboard with alerts, metrics, and information relevant to him. On the other hand, Prometheus management is more like a part of Kubernetes deployment. Therefore it is a decision of the individual team if Prometheus backend gets managed by the developer or operational part of a team.

	Prometheus	Sysdig Monitor
Instrumentation	Agent-per-pod	Syscalls
Data Types	JMX and StastD after configuration	JMX, StastD and Events
QL	Yes	No
Alerting	Build in AlertManager	GUI interface
Management	Part of Kubernetes deployment	Separate management for developers/operators

Table 5.1: Summary of key differences between Prometheus and Sysdig Monitor

To conclude, the main differences between Prometheus and Sysdig Monitor have displayed in table 5.1. Choosing Prometheus or Sysdig Monitor needs to be based on monitoring, user access, and data management requirements, and if you want to buy or build a solution. Therefore there is no generally better tool for container monitoring.

Chapter 6

Smoking Cessation Web Application

The practical implementation of technologies discussed in previous chapters will be built on the Smoking Cessation Web Application. It is an app, which is for smokers who decided to quit smoking. It provides a simulation of real-life therapy through everyday session's dialogues with a virtual therapist. The therapy is divided into two parts. The preparation part takes 10 days. During this time, the user still smokes and collects the motivation why he/she should stop. After 10 days of preparation, day D will come, when the user smokes the last cigarette. Afterward, the therapy follows for several weeks, while the virtual therapist tries to help the user to keep the motivation and strength for not picking up the cigarette through everyday short talks which last around 10 minutes.

Our smoking cessation web application is based on open-source logic-driven project Serafin developed by Inonit AS [16]. This application provides flexible building blocks and therefore it is very suitable for usage in web forms and questionnaires or e-learning programs. The core of the project contains generic features for everyday sessions such as text fields, multiple options questions, or buttons. Because of the separation of standards elements and the data which has to be filled out for concrete usage, it is very easy to change the context of sessions or to add new language to the application.

6.1 Logical Structure

For the purpose of this diploma thesis, a basic understatement of the logical structure of the application is needed. As mention above, the application simulates real-life therapy through every day sessions with the virtual therapist. Sessions are build from series of pages or other events like SMS, emails, or notifications. The core framework runs as a decision tree algorithm. Based on it each particular user path is distinguished.

The core decision tree consists of several crucial units:

- phase - the whole program is divided into two phases, the preparation and follow-up phase. This division of therapy was suggested and studied by Kulhanek in his paper [17];

- session - every day n th + 1 session is started, where n is the amount of already finished sessions by the user. Each session has its own decision sub-tree, which is responsible for rendering corresponding content. The sub-tree of session 1 can be seen in fig. 6.1, where additionally, compare to following sessions, registration of the user is required;
- node - represents actions made by admin such as rendering pages, sending emails or notifications, or starting a new session; and
- edge - connects different nodes and ensures progress in the decision tree. There are two types of edges. An automatic movement to the next node is happening when an edge is marked by a value pass, or conditional movement when an edge contains a boolean expression which has to be fulfilled to be able to move further down through this edge in the decision tree;

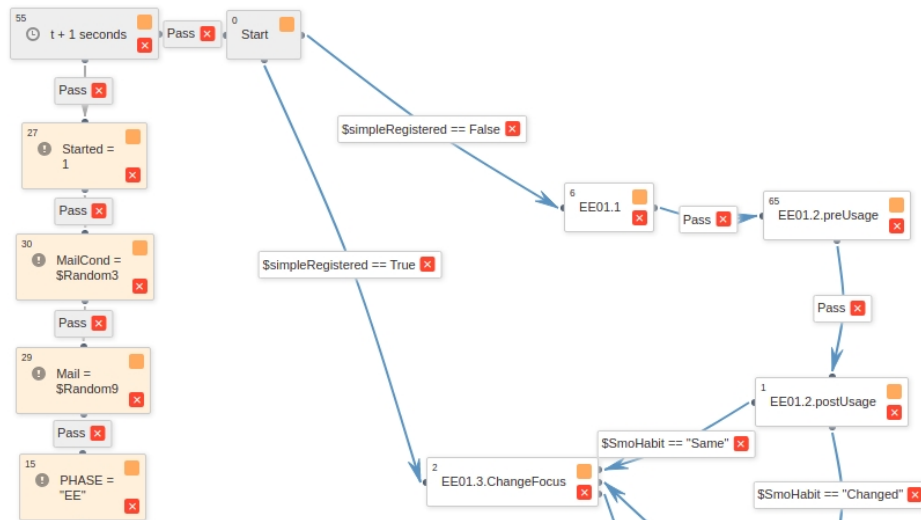


Figure 6.1: Part of the decision subtree of session 1. An orange node represents saving variable value of concrete user to the database, such as phase in which the user is or if he/she has already finished the session for the current day. Session 1 sub-tree also controls if the user is already registered. In case not, the registration process is started.

The program starts with the registration of a smoker. After that initial page is rendered, where basic data about the user are collected. Then a session starts and the user follows up its specific path in the decision tree, where the path is selected based on previously collected data. The session is built from a sequence of pages. After finishing the session for the corresponding day, the start of the next session is scheduled for the consecutive day. This is done by adding a task to the queue for scheduled execution. For the purpose of this thesis, it is important to know that trigger for a new day is scheduled for all the users at the same time and therefore this time is the most critical point in case of used amount of resources simultaneously since the rest of the

events started by users are spread out over the whole day. The whole therapy last around 2 months.

6.2 Technology

The computation server of the app is based on Django - a high-level Python Web framework. The server is connected to PostgreSQL database storage, where all the data about users and sessions are collected. The server unit is also connected to the queue, where scheduled tasks and short-delayed tasks are processed. These tasks are saved in key-value storage Redis, as can be seen in fig. 6.2.

Django is an open-source framework that enables rapid and secure development of websites. It takes care of user authentication, templates, content administration, management of passwords, and other basic security issues such as cross-site scripting or clickjacking. Because Django is easily scalable and it is possible to build all kinds of websites from content management systems to social networks, it is widely popular. Therefore also a broad range of plug-ins libraries is available. However, they are often not compatible with all the versions of Django, so in case of upgrading to a newer version of Django, some libraries have to be completely replaced with new ones.

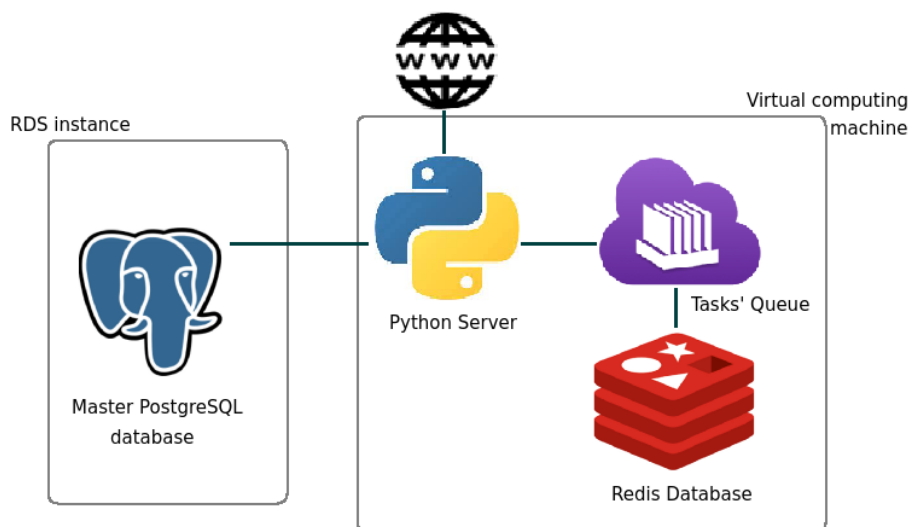


Figure 6.2: The architecture of the app. Grey rectangles represent individual instances running on the cloud server.

The app currently runs on the cloud computing server AWS. PostgreSQL database has its own RDS instance, which is an instance developed by Amazon especially for easy setup, operation, and scaling of a database in the cloud. Everything else, main computation unit, tasks' queue, and Redis database share one virtual computing machine instance. They run as separate virtual machines with their separated OS. This is not the best solution, as the

Chapter 7

Solution design and its implementation

The second version of the app described in chapter 6 is under development. The main reason for the new version was the upgrade of used technologies. The original version of the app is using Django 1.8 version, which is not supported since 2018, and python 2 which support ended in 2020. As part of the update, the new infrastructure is designed. To maintain, test, and deploy the application more easily, the Docker containerization approach was chosen. The basic design of 4 containers (PostgreSQL - main database, Redis - short term storage, Huey - tasks' queue, and App - the main computation unit) was formed by one of our colleagues in [20]. The natural follow-up is to use an orchestration platform to manage and monitor these containers, and to bring higher stability to the application. As the final step, we will design and implement a basic CI/CD pipeline for more comfortable deployment.

7.1 Tuning of the upgrade and containerization

The upgrade to a newer version of Django was not completed properly. The plug-in for the admin of the old version of the app was not compatible with newer versions of Django, which interfered with its basic functionality. Because the admin interface is used daily, we needed to find an alternative as quickly as possible. The requirements based on which we were choosing were compatibility with Python 3 and Django 3, timeliness of the project, and popularity among the IT community. Based on these conditions it was straightforward to use the Django-grappelli admin app, which is currently obviously the most popular admin package [21]. The already integrated new admin interface can be seen in fig. 7.1.

Additionally, to already build containers, we have formed one more PostgreSQL container for a supplementary secondary database. Creating a database container is a quite simple process because the basic images of different databases are available at the Docker hub, which is a repository for Docker images. We have extended the downloaded PostgreSQL image with volume, environment variables, and binding of ports according to our database.

Furthermore, we needed to secure that the migrations of the database will be created and applied at every build of the container. The solution for it was

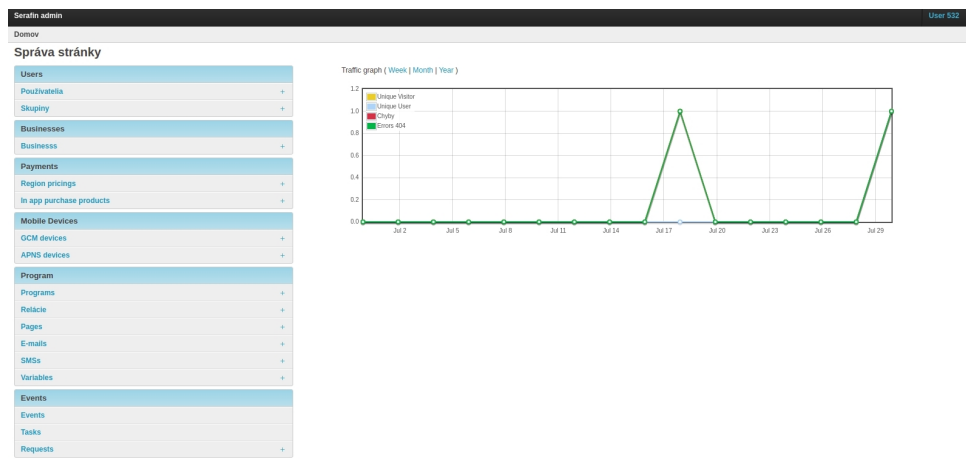


Figure 7.1: Part of new admin interface.

a little bit complex. We needed to apply migrations after creating an image but before starting a container. This way, we could be sure that changes in the schema of the database happened at the currently running container. However, docker allows only call single line command when starting a container. We have solved this problem by calling as a single line command part of a makefile, which contains instructing for creating and running migrations and starting a container at the proper port.

Finally, we have added the creation of static files and the installation of npm dependencies when building an image. Therefore no more manual work for the proper build of application images is required.

7.2 Cloud infrastructure management

7.2.1 AWS

The new version of the application is meant to eventually run, similarly to the old version, on the AWS cloud. Naturally, it makes sense to first set up cloud infrastructure, and then on top of it build orchestration. Since we are working on a life application, we need to set up a new infrastructure for a developer environment. Because it is a developer environment, the expected number of users is low. Therefore as the main server, we have chosen an instance with 2 vCPUs, 4 GiB of memory, and a rate of data transfer up to 5 Gigabit. In this instance, we run all the containers except PostgreSQL ones. The size of our database is around 250Mb, and further growth is expected because currently data about all the actions taken by users are being stored permanently. Therefore for database containers, we have chosen an RDS instance with 1 GiB of memory. As part of further security protection, we have set up possible incoming TCP traffic from the only main server IP address.

7.2.2 Swarm Orchestration

As we have introduced in chapter 3, currently there are two main orchestration platforms on the market suitable for Docker containerization. Kubernetes and Docker Swarm. When choosing a proper orchestration platform, it is important to consider mainly the complexity of the project, traffic load, and human resources. Our application is divided into 5 containers. It is currently actively used by up to a couple of hundred unique users per day. Based on the discussion which we made in 3.3 we see that one of the biggest weaknesses of Swarm, which is relevant to our situation, is scaling only on demand. However, even when the number of users would be multiple times bigger, it would still not required horizontal auto-scaling, which Kubernetes is offering. Swarm offers to scale on demand which is enough for projects like ours, in which the number of users is growing steadily. We also concluded that Swarm is more suitable for small projects when there is no high load of traffic. It is also much simpler to use it because Swarm uses CLI of Docker with which we are already familiar. Also, there is no manual configuration needed when starting a new cluster. Because of all these advantages, we decided that Docker Swarm is a better fit than Kubernetes in this situation.

To create a Swarm cluster, Swarm mode has to be initiated on the manager node. In our scenario, we have created a Swarm Cluster consisting of one manager node. On this node, we run one stack consisting of 3 services relevant to our application - main computation unit, tasks' queue, and Redis database. All of our services are running in replicated mode, which means that an exact number of replicas was created regardless of the number of nodes in the cluster. As can be seen in fig 7.3 each of the services is having exactly one replica. However, in case it will be needed, services can be scale up easily with a single command separately or simultaneously through changing a configuration file and redeploying.

In the future, it is expected that more than one replica will be needed, especially when considering the production environment. More replicas are helping to reach the maximum uptime and better stability. However, this is more a business decision as running more services requires finances. In the case of having multiple nodes in the cluster, proper ports need to be opened. It allows new nodes to connect to the Swarm cluster and communicating with each other.

```

└─$ sudo docker service ls
ID                NAME                MODE                REPLICAS  IMAGE
0sfikuut5oq8     portainer_portainer replicated          1/1       portainer:latest
45dxi6zi5nv      serafin_app         replicated          1/1       serafin_app:latest
wq0nlay0b7ts     serafin_huey        replicated          1/1       serafin_huey:latest
44vg5xyjb9lu     serafin_redis       replicated          1/1       redis:5.0

```

Figure 7.2: List of services deployed in the Swarm cluster. We have deployed 3 services regardless of the application. The fourth service is the UI management tool Portainer, which runs as a single container.

As mention in 3.4, Swarm does not come with integrated UI management.

Therefore on top of Swarm, we had to implement a UI interface. We have chosen the Portainer container management tool because it is lightweight, fully developed, and currently the most popular management tool. It runs on a single container. We have deployed a Portainer container in a separated stack on our manager node to achieve isolation between applications. It runs as a single replica as there is no need to have multiple replicas of Portainer since, in case of failure, Portainer can be started and connected to our application within few minutes.

7.3 GitLab CI/CD

We are hosting our shared repository on GitLab, which is one of the most popular end-to-end developer platforms. Except repository hosting it provides various tools for the software development cycle, including build-in continuous integration tool GitLab-ci. Because our application already uses Gitlab as a shared repository, GitLab-ci is a fast and easy solution for the implementation of the CI/CD pipeline. To set up a pipeline no third-party tools are required. Instead, a YAML file in the root directory of the project has to be created inside of which pipeline is defined. Every time a code is pushed into the Gitlab repository, it automatically looks and executes a gitlab-ci.yml file.

Every gitlab-ci.yml file consists of stages and jobs. Jobs define what is gonna happen and the stages when the jobs are gonna be executed. Each job is executed by a runner, which needs to be connected to the repository. If we have more than one runner available, multiple jobs can run simultaneously, speeding up the process significantly.

We have designed a pipeline for a newly set up environment on AWS, which we have described in the previous section 7.2.1. The goal was to introduce such a solution of CI/CD pipeline that all direct human manipulation with the server would be eliminated. The part of the configuration file of suggested pipeline is displayed in listing 7.1. The pipeline gets executed only when new code is pushed to the master branch. The master branch is protected, and therefore it is only possible to merge into it after a review. This increases the quality of code and makes sure that non-valid codes do not get deployed to any environment. Thanks to tags, we can easily distinguish which commits should run through CI/CD pipeline.

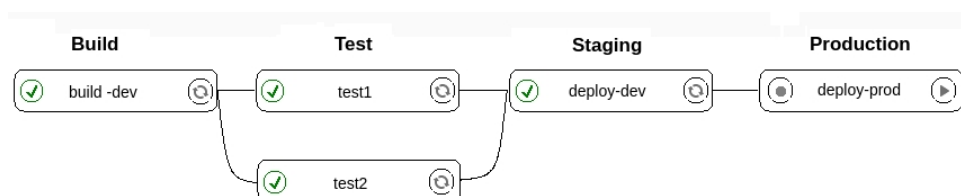


Figure 7.3: The diagram of the designed CI/CD pipeline, with parallel work during the testing stage. The first three stages have passed automatically, the fourth stage is still waiting for a manual trigger.

The pipeline currently consists of four stages. The build, test, staging and production stage. The build stage controls if the application gets build without any errors and if it is compatible with its libraries. A script with defined containers is executed, as can be seen on line 14 in listing 7.1. During test stage, the script with automatic tests gets called. We have designed a parallel testing to speed up the process. To make this possible we needed to register a second GitLab runner. However, currently, there are no scripts written for testing, and therefore we have only prepared the configuration for testing stage. Staging stage is automatically deploying an application to the developer server, which location is defined on line 19.

```

1 stages:
2   - build
3   - test
4   - staging
5   - production
6
7 build-dev:
8   stage: build
9   tags:
10    - dev
11  only:
12    - master
13  script:
14    - docker-compose -f docker-compose.yml build
15
16 deploy-dev:
17   ... #stage, tags, and branch are defined
18  variables:
19    DOMAIN: ec2-34-247-48-58.eu-west-1.compute.amazonaws.com
20    DJANGO_SETTINGS_MODULE: serafin.settings_aws_dev
21    DJANGO_DEBUG: "True"
22
23
24 deploy-prod:
25   ...
26   when: manual #starts via manual interaction

```

Listing 7.1: Part of the pipeline configuration file. Because of space restriction, the complete file with the test stage and the rest of the file can be found in files attached to this thesis.

Because we have set up only one environment, the CI/CD pipeline is fully automated only for the developer environment for now. However, we have constructed the configuration file with future expectations of multiple environments. After a review and confirmation of proper functionality in the developer environment, the newest version can be manually deployed to production using a GitLab GUI, once the domain of production server is known.

7.4 Prometheus

To make sure that our Swarm infrastructure is healthy and as efficient as possible, we need to use a container monitoring tool. The basic features of performance monitoring have been discussed in chapter 5, where we also compared the currently two most used tools. For monitoring our application, we have decided to use Prometheus.

Our current infrastructure of application consists of three services, two databases, and a Portainer UI. Because of the size of the infrastructure, it is enough that one host serves the whole application. When using a Swarm cluster, in our case, we want to have one replica of each container at each host. When monitoring the application, we need to obtain information about individual services and containers. Sysdig Monitor uses Syscalls infrastructure therefore it is not a suitable solution for use. On the other hand, Prometheus uses the agent-per-pod method as well as having its own query language. Also, the future plan is to eventually have full container monitoring with alerting which Prometheus is offering.

We have set up Prometheus on our manager node of Swarm. We have decided to use Prometheus for monitoring the Docker daemon and its container. To be able to monitor the Swarm cluster, Prometheus needs to have access to the Docker socket, and the Docker daemon has to expose its metrics to some port. For monitoring containers, we have additionally deployed Cadvisor as a separate service. Cadvisor is a daemon that exposes running container metrics. These metrics are gonna be sent to Prometheus, where they can be further processed.

The Prometheus behavior is determined by a configuration file. The configuration file consists of a global and scrape part. The global part defines parameters that refer to all other configuration contexts, e.g. how often to scrape target. The scraping part specifies jobs and their parameters. Job is a collection of instances with the same purpose. In jobs targets, that are supposed to be scrape by Prometheus are defined.

```
1 scrape_configs:
2
3   - job_name: 'docker'
4     static_configs:
5       - targets: ['localhost:9323']
6
7   - job_name: 'Docker_Containers'
8     static_configs:
9       - targets: ['localhost:8080']
10
11  - job_name: 'prometheus'
12    static_configs:
13      - targets: ['localhost:9090']
```

Listing 7.2: Part of the Prometheus configuration file.

As it can be seen in code listing 7.2, we have defined three different targets in our configuration file. Firstly, we are monitoring the Docker daemon which we have set up to export his metric to port 9323. The second job is collecting metrics of individual containers. These metrics cannot be collect by Prometheus alone. As described before we have deployed Cadvisor as an intermediate layer between Prometheus and containers. Container metrics collected by Cadvisor are sent to port 8080 and further processed by Prometheus. As last we have decided to monitor also the behavior of Prometheus alone.

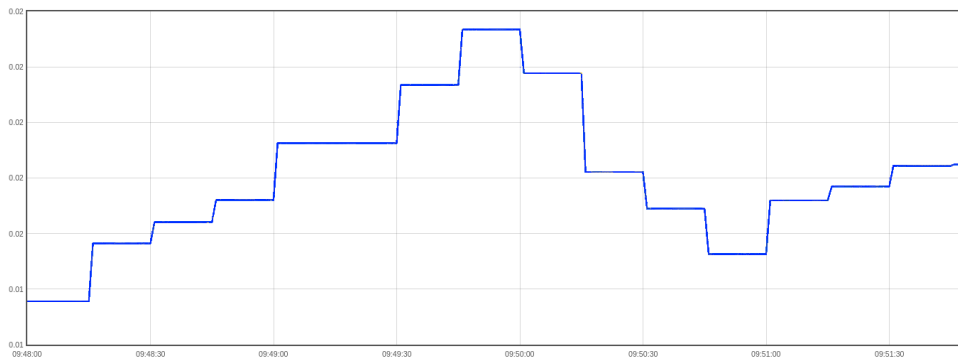


Figure 7.4: CPU usage of one of our containers is displayed in Prometheus.

Prometheus provides a basic UI. By using a Prometheus querying language, various metrics are displayed in either table form or graph form as we did in fig. 7.4, where we obtain information about the usage of the CPU of one of our containers. Additionally, a better graphical interface named Grafana can be added to Prometheus. Also, AlertManager for sending alerts to various information canals such as slack or email can be attached to Prometheus.

Chapter 8

Experiments and Results

In this chapter, we will show the improvement, which was potentially reached by rebuilding the infrastructure of the application. We will focus on the performance and stability of the application, which should be improved because of containerization and orchestration tools. We will also examine how much simpler and faster is deployment thanks to CI/CD pipeline.

8.1 Performance tests

The orchestration of infrastructure affects the stability and efficiency of the application. We will compare the performance of the old and new versions of the application. The test will be performed with help of JMeter. JMeter is open-source software for load test functional behavior and measure performance.

We will be sending HTTP get requests to the first page of therapy. The performance test scenario will use the following parameters:

- Number of users - 2000
- Ramp-Up period - 4000s
- Loop count - 10

Ramp-Up period determines the delay between two users. The delay period is calculate Users/Ramp-Up , so in our case every 0.5 second a new user is started.

In fig. 8.1 we can see the behavior of the old version of the application during the test. However, the first look at the graph does not tell that much about the performance of the application. For this reason, we need to go to bigger details, especially have a look at the specific values. The most important parameters are throughput, deviation, and the latest sample. The throughput is the load processed during the unit of time. In our case, it is a little bit more than 295 per minute, which means that the application can handle around 5 users every second. The deviation shows us how close the values are to the average. The lower deviation, the better because the application is performing more constantly, without unpredictable behavior.

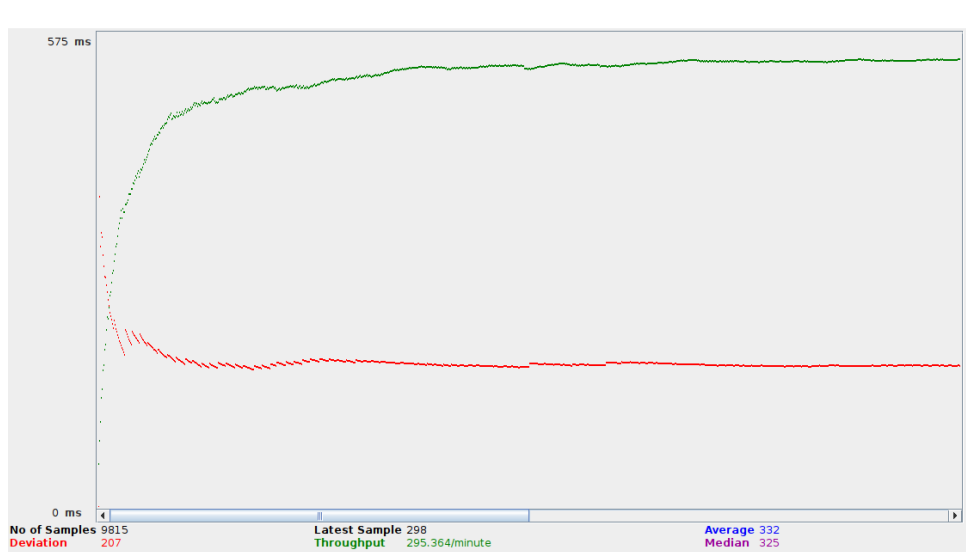


Figure 8.1: The result of performance test of the old version of the application.

In the case of the old version, the deviation is 207. The latest sample is the response time for the last requested URL in milliseconds. In our case, it is 298 ms. However, we can see that the y-axis goes up to 575ms, which was the worst response time. The throughput and deviation are fairly constant during the test, so the behavior of the application is quite consistent.



Figure 8.2: The result of performance test of the new version of the application.

In comparison, fig. 8.3 shows the performance of the new version of the application. The throughput is around 604 per minute, which means that the application can serve 10 users per second compared to 5 users which the old application was able to handle. The deviation is 12, which is quite low. Therefore the performance of the application is consistent, which we can also see from the graph where the green line is basically constant. In the beginning, the throughput is growing until it reaches its saturation point. Compared to the graph in fig. 8.1, the saturation was reached much faster

because the users were handled quicker. The latest sample is 19 ms, with a maximum of 21ms. This is lower compare to 207ms in the old version.

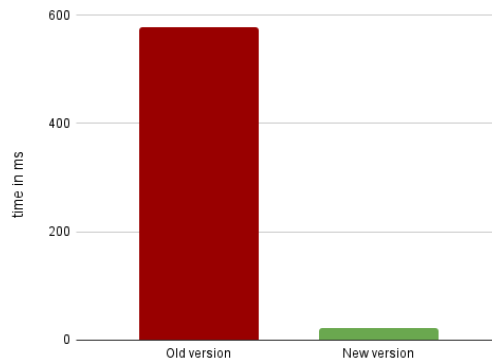


Figure 8.3: Comparison of time response of an old and new version of the application.

In the second test on performance, we will have a closer look at the CPU usage of containers. Because we will perform requests to part of the website in which reading from the database is required, the two countries which are gonna be affected are the main computation unit and database. This time we will be sending via JMeter 40 requests with a 1-second delay between users. Therefore the test will last 40 seconds. The output will be measured in Portainer, a UI for Docker Swarm. Except for CPU statistics, Portainer gives also information about memory, I/O, network usage for each container. However, these statistics are only real-time, no historic data are available.

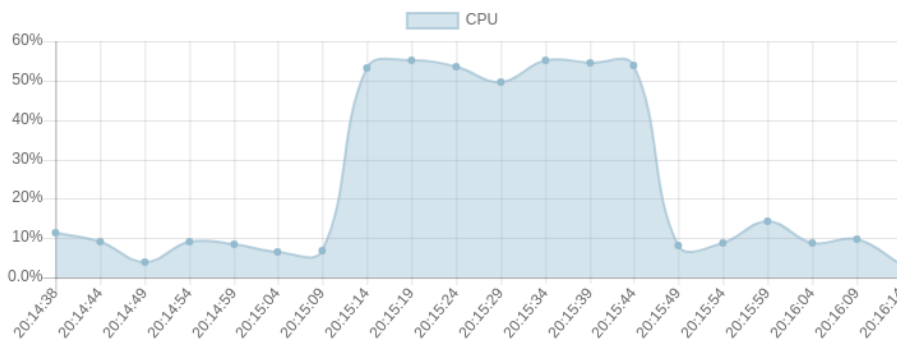


Figure 8.4: The CPU usage of main computation unit

In fig. 8.4 we see CPU usage of the main computation unit during the test. At the beginning of the test, we see a growth from approximately 8% of CPU usage to 55% in five seconds. This is the beginning of the test. After the end of the test, we see a steep decline of CPU usage, to approximately 8% around which the CPU usage is oscillating.

In fig. 8.5 we see CPU usage of the database during the test. However,

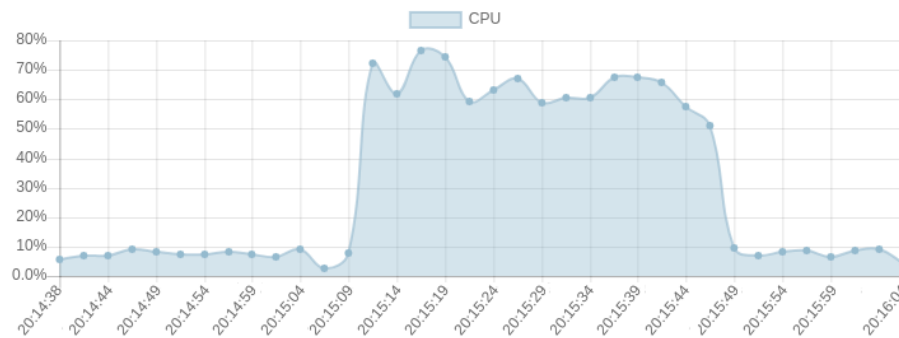


Figure 8.5: The CPU usage of database while read request were send to it

compared to the main computation, the CPU usage grows to 60%, in some parts even up to 70%. When we tried to run the same test, but we changed the delay between users from 1 second to 0.5, we have reached the CPU usage of more than 100%, so there is a place for improvement in the database organization. However, the overall high usage of CPU by both containers while having a relatively small amount of users is probably caused by the fact that at the moment of testing, except for app containers, there were also running containers regarding the container monitoring and Portainer UI at the same machine. Overall we had 8 containers between which the resources had to be shared.

8.2 Deployment improvement

Automated deployment through GitLab CI/CD pipeline is minimizing the manual interaction and making the process of deployment as efficient as possible. To capture how efficient is to deploy through CI/CD pipeline we measured two variables: time needed for deployment and the number of manual steps which has to be performed. Tests were taken only on currently used stages, which are the build and the staging stage. The results of both tests can be seen in fig. 8.6.

First, we measured the time of deployment. We were comparing the time of deployment when no changes in migrations of database and in Docker file were happening. That means that the build of the container was performed fast because of the layer system in Docker. We have executed both manual and automated deployment 5 times and took the mean of these data. The time of the automated GitLab CI/CD deployment was measured by the GitLab alone and the mean of 5 tries was 8 seconds. In the comparison, the manual deployment took 52 seconds. However, this time can be significantly longer as we had set up excellent conditions for testing. For example, the commands needed for execution were used as the last ones in the terminal, so it was possible to list them fast from the history, which gave us significant

time-saving.

The second test which we did was a comparison of manual steps needed for deployment. When using GitLab CI/CD pipeline, only one command needs to be performed, committing to GitLab. In the case of manual deployment, extra steps such as connecting to a server, changes between directories, creation of migration, migrating changes to the database, and deployment need to be executed. Overall, in ideal conditions, we counted 9 steps that are necessary to perform.

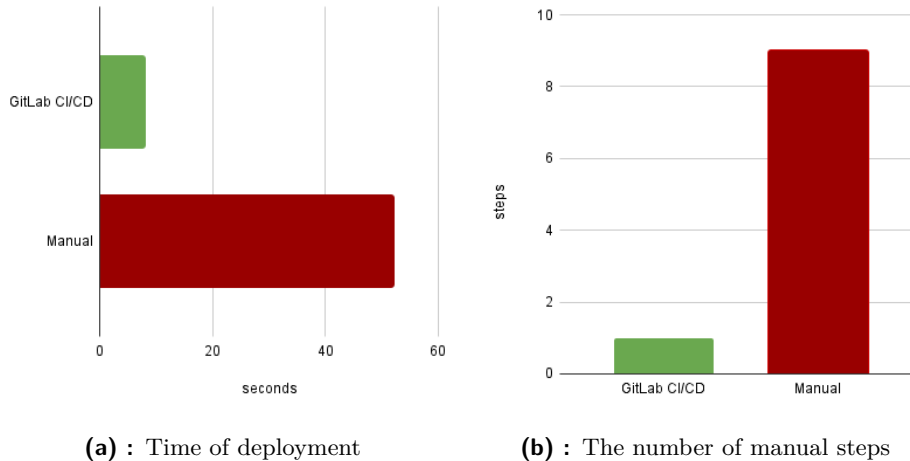


Figure 8.6: Comparison of the efficiency CI/CD pipeline and manual deployment

To conclude, the efficiency gap will grow even further when more stages are gonna be included in the pipeline. Additionally, testing and deployment to different environments can be implemented. In this case, manual deployments become extremely inefficient because we will have to execute commands on multiple hosts which complicates the whole process of deployment. Also in the future it will be interesting to observe if the time between deployment of two versions is shortened. However, this will show later in real life usage of designed CI/CD pipeline.



Chapter 9

Conclusion

The target of this thesis was to improve the stability and efficiency of the smoking cessation web application using containerization and orchestration tools. This is still a quite new area of computer science. Therefore at first orchestrating an infrastructure can be quite challenging for people who are not familiar with it. Naturally, after orchestration, the next step is to implement CI/CD pipeline to automatize deployment.

While getting familiar with the application, we worked on a new admin interface and adjusting docker files which are creating containers. Afterward, we have set up the application to cloud service AWS, where we have implemented the solution for orchestration of the app using the Docker Swarm platform. The cluster currently consists of one node as it is enough for the current traffic load however, it is easily scalable. We have also added a UI interface. Therefore the creation of clusters, containers, rollbacks, and all other features offered by Swarm can be performed easily without deep insight.

In the second part of our work, we were focusing on the creation of a CI/CD pipeline. For this purpose, we chose GitLab, where we were already hosting our repository. The suggested pipeline consists of four stages, even though only build and staging stages are only currently needed. Additionally, we have prepared a test stage that runs parallel on two runners to make the testing stage faster. Also, we added the production stage, which is triggered manually. Next, we dived into the topic of container monitoring which brings various challenges because of the way how containers were designed. Out of multiple platforms which are on the market, we chose Prometheus. We have implemented Prometheus with cAdvisor as an intermediate layer to obtain statistics about individual containers.

Finally, we performed few tests. Performance tests confirmed the higher efficiency and stability compare to the old version. However, more impressive were deployment tests where we compared manual deployment with an automated one. The complexity of deployment was extremely reduced basically to only one command. We were also able to deploy our application in 8 seconds when the docker image was taken from the cache. Therefore we can overall conclude that we had successfully designed and implemented the CI/CD pipeline, which was the main purpose of this thesis.

■ 9.1 Future work

The application still needs some further tuning. The test scenarios for the test stage need to be written and properly fitted into the designed pipeline. The production stage is designed to be executed manually, however, environments variables and scripts need to be set up properly once the production environment is up and running. When looking at container monitoring, additionally an AlertManager can be added, which would send notifications to various communication canals in case of undesirable behavior.

Also, we have found out that the database is quite large, around 500MB. This is caused mainly because we keep information about accessing a single page from every user. This information may be valuable only for a couple of hours, maximum days, and therefore there is no reason for storing them permanently. As we saw in our test scenarios the CPU usage of the database container grows quite fast, which might be partly caused by the size of the database.

Appendix A

Bibliography

- [1] A. Khan, "Key Characteristics of a Container Orchestration Platform to Enable a Modern Application," in *IEEE Cloud Computing*, vol. 4, no. 5, pp. 42-48, September/October 2017, doi: 10.1109/MCC.2017.4250933.
- [2] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. "Xen and the art of virtualization," *SIGOPS Oper. Syst. Rev.* 37, 5 (December 2003), 164–177. DOI:<https://doi.org/10.1145/1165389.945462>
- [3] Boxdell, D. (2018, March 15) Containerization: The need to Know. [Blog post]. Retrieved from <https://www.armor.com/resources/blog/containerization-the-need-to-know/>
- [4] RedHat, (2020, January 16) Containers vs VMs. Retrieved from <https://www.redhat.com/en/topics/containers/containers-vs-vms>
- [5] Collabnix, (2017, December 13) Docker vs Container. Retrieved from <https://dockerlabs.collabnix.com/beginners/docker/docker-vs-container.html>
- [6] S. A. I. B. S. Arachchi and I. Perera, "Continuous Integration and Continuous Delivery Pipeline Automation for Agile Software Project Management," 2018 Moratuwa Engineering Research Conference (MERCon), 2018, pp. 156-161, doi: 10.1109/MERCon.2018.8421965
- [7] Choudhury, B. R. (2019). A modern CI/CD pipeline on pure [White paper] Pure Storage. <https://www.purestorage.com/content/dam/pdf/en/white-papers/wp-a-modern-ci-cd-pipeline.pdf>.
- [8] Fox A, Patterson D. *Engineering Software as a Service: An Agile Approach Using Cloud Computing*. StrawberryCanyon LLC, 2014.
- [9] Portainer, (2021, April 4) Application Management. Retrieved from <https://www.portainer.io/products/application-management>

- [10] Docker, (2016, July 27) How services work. Retrieved from <https://docs.docker.com/engine/swarm/how-swarm-mode-works/services>
- [11] Path, S. (2020, July 26) CI/CD with Jenkins pipeline Nodejs into K8S (Part-2). Retrieved from <https://sampath5898ch.medium.com/?p=ad01eb75bba9>
- [12] Canonical Ltd. (2021, 9 August) MicroK8s Release notes. Retrieved from <https://microk8s.io/docs/release-notes>
- [13] Silva, Vitor Kirikova, Marite Alksnis, Gundars. (2018). Containers for Virtualization: An Overview. Applied Computer Systems. 23. 21-27. 10.2478/acss-2018-0003.
- [14] Prodan, S. (2016, Octobere 6) A monitoring solution for Docker hosts, containers and containerized services. Retrieved from <https://stefanprodan.com/2016/a-monitoring-solution-for-docker-hosts-containers-and-containerized-services/>
- [15] Casalicchio, Emiliano Perciballi, Vanessa. (2017). Auto-Scaling of Containers: The Impact of Relative and Absolute Metrics. 10.1109/FAS-W.2017.149.
- [16] INONIT.NO, (2018) Serafin framework. Retrieved from <https://github.com/inonit/serafin>
- [17] KULHÁNEK, A. et al. eHealth Intervention for Smoking Cessation for Czech Tobacco Smokers: Pilot Study of User Acceptance. 2018, Page(s): 81–85
- [18] Böhm, Sebastian Wirtz, Guido. (2021). Profiling Lightweight Container Platforms: MicroK8s and K3s in Comparison to Kubernetes.
- [19] DOCKER, (2018, August 7) What is a container. Retrieved from <https://www.docker.com/resources/what-container>
- [20] J. Trmal, Optimization of server solution and performance measurement. 2020. Czech Technical University in Prague. Computing and Information Centre. Retrieved from <https://dspace.cvut.cz/bitstream/handle/10467/90234/F3-DP-2020-Trmal-Jakub-thesis-trmal.pdf?sequence=-1&isAllowed=y>
- [21] Django, (2021, August 11) Admin interface. Retrieved from <https://djangopackages.org/grids/g/admin-interface/>
- [22] J. S. Sanz, E. Carter, and K. Anderson. Running Containers in Production for dummies - Special Edition. John Wiley Sons, Inc. 111 River St. Hoboken, New Jersey. 2018. ISBN 978-1-119-52105-1
- [23] GitLab, (2016, July 14) CI/CD pipelines. Retrieved from <https://docs.gitlab.com/ee/ci/pipelines/>