Czech Technical University in Prague

Faculty of Electrical Engineering

Department of Computer Science and Engineering

Master's Thesis

Optimization of server solution and performance measurement

Jakub Trmal

Supervisor: doc. Ing. Daniel Novák, Ph.D.

Study Programme: Open Informatics, Master

Field of Study: Software Engineering

August 11, 2020

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Trmal Jakub**	Personal ID number:	**457118**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Computer Science**		
Study program:	**Open Informatics**		
Specialisation:	**Software Engineering**		

II. Master’s thesis details

Master’s thesis title in English:

Optimization of server solution and performance measurement

Master’s thesis title in Czech:

Optimization of server solution and performance measurement

Guidelines:

The student will study and analyze the existing server-side application. The server needs to update all used technologies to new supported versions. It also needs to implement some optimization of the core engine to run more efficiently. The server will adequately set up for Continuous Integration and Continuous Delivery tools to simplify the deployment process.
1) Study and understand the existing project.
2) Update technologies of server project to new supported versions.
3) Apply some optimization technics to the Decision Tree algorithm used by the server so it will run more efficiently.
4) Study Docker container technologies and use these technologies to deploy this server on the testing and production platform. Configure Continuous Integration and Continuous Delivery for this project. The optimization will be tested by comparing the runtime performance on the previous and the current implementation.
5) Make sure every communication with clients (e. g. mobile applications, frontend application, or external clients) stay in good condition.

Bibliography / sources:

1] H. Brendryen, P. Kraft, and H. Schaalma, “Looking Inside the Black Box: Using Intervention Mapping to Describe the Development
of the Automated Smoking Cessation Intervention ‘Happy Ending’,” The Journal of Smoking Cessation, vol. 5, no. 1, pp. 29–56, Jun.
2010.
[2] H. Brendryen, F. Drozd, and P. Kraft, “A digital smoking cessation program delivered through internet and cell phone without nicotine replacement (happy ending): randomized controlled trial.,” Journal of medical Internet research, vol. 10, no. 5, p. e51, Jan. 2008.
[3] Kulhánek A., Gabrhelík R., Novák D. &amp; Brendren H. (2018). eHealth intervention for smoking cessation for Czech tobacco smokers:
Pilot study of user acceptance. Adiktologie, 18(2).

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: **17.02.2020** Deadline for master's thesis submission: **14.08.2020**

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 his 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 all colleagues from the project for providing an opportunity to realize this work, discussion during the creation process and generally for the time spent on implementation. Great gratitude also belongs to my girlfriend and family for providing the environment to work on this thesis. Thank you for your overall support during my studies.

Declaration

Abstract

In this diploma thesis we apply modern software engineering techniques and concepts to enhance existing web application-based projects. We ensure latest software equipment for the project, afterwards we make neccesary modifications to prepare the setup for continuous integration and continuous delivery for the purpose of a more comfortable development process and prevention of automatically detectable failures. A well-designed project can significantly profit from the entire potential of modern cloud computing platforms and provide simplification of the deployment procedure at the same time. We also introduce optimization techniques, whose aim is to improve user experience and reduce computation time of internal data processing as well. Some of the optimization techniques are chosen to directly improve this particular application performance, besides some of the techniques are more universal for general usage during software performance optimization. The most suitable techniques are implemented in practical part for illustration.

Keywords: *Python web server, web server optimization, continuous integration, continuous delivery, update strategies, Docker containerization, microservices architecture*

Abstrakt

V této diplomové práci se zaměříme na aplikaci současných technik a konceptů softwarového inzenýrství, které nám pomohou modernizovat existující webovou aplikaci. V první řadě se zaměříme na aktualizaci technologií, na kterých je projekt postaven. Následně projekt připravíme na plné využití průběžné integrace a nasazení, čímž získáme větší komfort při vývoji ale také automatické odhalení chyb vzniklých implementací nových úprav. Dobře navržený a modernizovaný projekt může zjednodušit celý proces nasazení aplikace do provozu a zároveň může výrazně profitovat z možností, které dokáže nabídnout prostředí cloudu. Dále v práci teoreticky prezentujeme možné výkonové optimalizace dané aplikace, jejichž hlavním cílem je zlepšit uživatelský zážitek ale také redukovat čas běhu výpočetních úloh na pozadí. Některé z technik jsou přímo volené pro optimalizaci konkrétně tohoto projektu, ale jiné mohou být inspirací pro obecné výkonové optimalizace softwarového projektu. Konkrétně využitelné techniky implementujeme v praktické části práce.

Klíčová slova: *Python web server, web server optimalizace, průběžná integrace, průběžné nasazování, aktualizační strategie, Docker kontejnerizace, architektura mikroservis*

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this diploma thesis, we will introduce some techniques in workflow, which are used to update the server framework and improve its runtime or memory consumption. Optimization is also closely related to developers. If the automation of monotonous but necessary steps in the development process is well set, the developer can be more focused on his creative part of the job. A good set-up of repeating processes, like building application, tests or deployment, can save time. Not only the time spent by the developer to go through this pipeline every day, often multiple times, but also the time to initiate a new member to the team. A good setup can also save money which is closely related to spent development time. Last but not least, the automation can discover a significant amount of bugs which cannot always be evident.

As the model for implementation and measurements, we will use the server part of the application which provides virtual therapists to help people quit smoking [12]. The second chapter will introduce initial back end side condition, architecture and used technologies. In the third chapter, we will take a more sophisticated insight into technologies, choice of the most useful update strategy and overall preparation for a new architecture, which will be mostly about converting the monolithic application to distributed microservices. Afterwards, we focus on the automatic deployment process in the context of new microservices, which are based on containerization. For this purpose, it is suitable to design the pipeline connected to GitLab VCS, which can help us to simplify the delivery process. Finally, if we have a working application deployed as microservices, we can focus on optimization. The optimization can save many resources, but we should be cautious, and all changes should be moderate. In the end, we focus on experimental measurements, which should confirm or disprove the changes and compare the application performance before and after.

1.1 Goals

As the first, it is essential to gather all possible information about the web application concept and understand general intent on how it was created. If we want to optimize the whole core system, it is necessary to fully understand the state in which the system is now and define the target state the client wants to reach. Firstly, we set the acceptance criteria and evaluation methods which help us to measure improvement or deterioration after some optimization is applied. We want to present the train of thought in the course of the whole process and discuss them with the client.

In this case, we can use the belief that the base web application was created as a general framework and the client runs a particular implementation which is rarely changing. At the beginning and development phase of this project, it was a huge benefit to work with the more generally based framework. It brings us the advantage of the flexibility and fast production of new features, but nowadays, the project is more stable and ascertained as functional. Due to this fact, we can start by removing some unnecessary general parts and speed up the application, with less memory consumption and more stability.

The second goal is to prepare the web application for automated deployment using containers, which is a no less important part of software engineering work.

The last challenge is the optimization of the whole application runtime. We try to find the critical parts and modify them to achieve better performance. If this section is thriving, we can save the computational time, and it can help the application to handle a more significant load.

Chapter 2

Technologies and Architecture

For the practical implementation, we use server application for a logic-driven opensource project Serafin developed by Inonit.no [11]. This application is well designed for general usage in e-learning programs, sophisticated web forms and other condition-based web applications, but for a fixed domain usage, it is superfluously universal. As we mentioned in the chapter 1 Introduction, the web application uses a virtual therapist who conducts a dialogue with the smoker and who is a companion for him in the whole program. The core framework uses predefined generic parts in dialogues such as multiple choices, text fields, images, buttons, et cetera. All these standard elements, which are well known from web designing, has specific data structure reusable to fill the elements with final data. For instance, data for language mutations text fields which we can store separately from core application data. To provide new language is uncomplicated due to this separation, and it can also be a good optimization opportunity.

2.1 Web application concept

In this section, we will briefly go through the application concept and main ideas which we need to understand. We are able to build a full addiction treatment plan which is spread out over several days duration, based on the idea behind the framework. The core of the framework uses a decision tree algorithm. The representation model for this algorithm has an ordinary tree form and consists of two main parts:

Definition 1. *nodes* - in this framework representing possible actions the system admin can apply. For instance, send an email to the smoker, render the page with information, display push notification, et cetera.

Definition 2. *edges* - connecting nodes as expressions. If the expression is evaluated as `true`, then this edge is used and the following action is triggered. In the case of `false` evaluation, the edges are skipped by the algorithm.

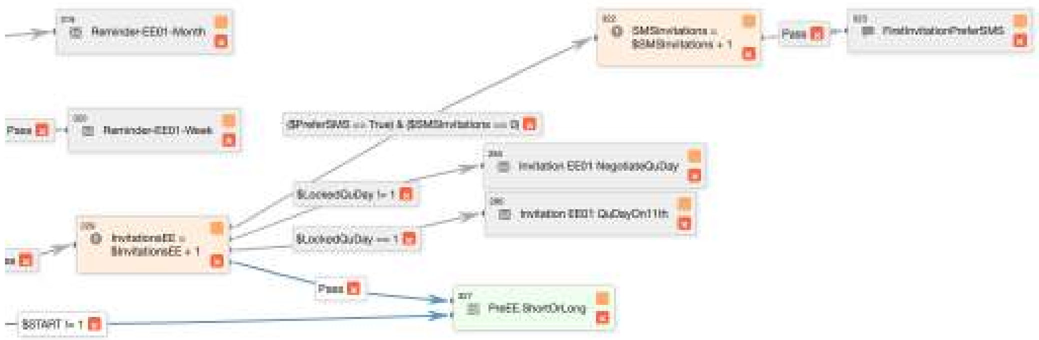

Reminder-EE01-Month
Pass
Reminder-EE01-Week
$PreferSMS == True & $SMSInvitations == 0
SMSInvitations = $SMSInvitations + 1
Pass
FirstInvitationPreferSMS
Invitation EE01 NegotiateQuDay
$LockedQuDay != 1
Invitation EE01 QuDayOn11th
InvitationsEE = $InvitationsEE + 1
$LockedQuDay == 1
Pass
PreEE ShortOrLong
$START != 1

Figure 2.1: User interface of Decision Tree editor. Part of the Decision Tree. As we can see, there are two types of edges, the first is evaluating some expression, and the second labelled "pass" means the direct transition to the next node. The orange nodes assign a value and grey nodes representing email or notification.

Definition 3. *phase* - the plan consists of two phases: preparation and follow-up, which Kulhánek presents in his paper [12].

Definition 4. *session* - the part of the decision tree defined by a current program. In a session we have united subtree belonging to the nth day, which consists from a set of actions.

The system is designed to be able to control the whole program with two phases in approximately one month. The entry point to the system for a smoker is registration. After registration is an information form for the initial setup of user variables. At the next step, the algorithm starts on user setup, which begins with edge evaluation and triggering a set of actions one by one in order of successful edges. Every day, after the session has completed the system schedule task for the next day. From the technical perspective, the job is added to the queue owned by a workers' pool for scheduled execution. Each task executes when the scheduled start time is reached. The crucial thing is how the smoker behaves at the nodes;

his actions are saved to his account and afterwards define his state onwards. This situation occurs when the algorithm reaches the orange node shown in figure 2.1 above. The decision is saved to the variable, and it is volatile, meaning the decision can be updated later and change the behaviour in the decision tree, as the next day always starts from the root. The edges are just read-only parts, compared to the nodes, evaluating the user state as binary formula into `true` or `false` result.

2.2 Technological background

From a technological perspective, we have a relatively compact server side. The web application is a typical illustration of monolithic architecture which means the functionality is concentrated in one place with the entire front end, back end and other utilities. This architecture has many benefits not only for the small application, but it leads the development to one big codebase, and it is not so flexible. If the system and performance requirements increase, it may be better to think about different architecture. For instance, nowadays, many applications are moving to Cloud and using famous microservice architecture.

Before we consider migrating this web application to the new architecture, we can discuss the current state. As a core computation unit, we have a Python-based server using web application framework Django[1]. The server is handling connection to persistent storage Postgres, which is a typical relational database using SQL—the following necessary link to the worker pool, which is fully concurrent queue processing unit. The primary function of this worker pool is to handle scheduled tasks and other short-time delayed tasks. This component is using in-memory key-value storage Redis as we can see in figure 2.2.

For the deployment, we are running a dedicated private server which provides resources for LXC[2] virtualization. Each development, testing or production environment has its own virtual machine with separate OS hosted on one physical machine. This may not be the best approach in case we want to increase production performance. In general, the infrastructure of virtual machines must have a well-set resource management strictly separating development and production. If the developer deploys a new feature to the development virtual

[1]https://www.djangoproject.com/

[2]https://linuxcontainers.org/

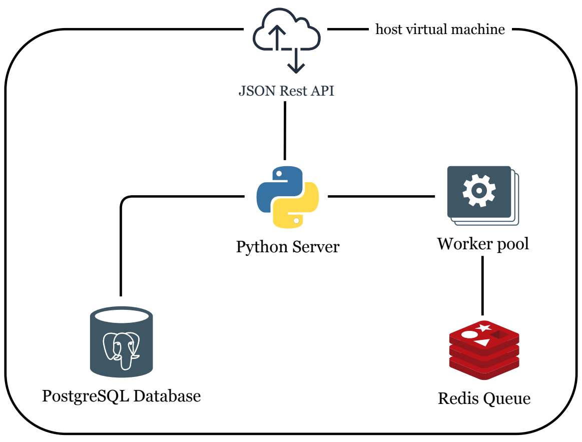


Figure 2.2: Monolithic back end architecture, which is hosted on virtual private server.

machine, which exhausts CPU or memory resources by mistake, it can affect production. From the optimization view, we can consider applying some slightly different isolation attitude - containerization. Nowadays, container-based virtualization is rapidly evolving, and it also has many additional framework possibilities. For instance, in good cooperation with containers, we can use various kinds of resource management tools, which provide almost autonomous control over the allocated CPU or memory.

The back end provides JSON Rest API, which is a pretty common standard of communication interface provided for clients like mobile devices or desktop stations. In fact, this protocol can be fast enough, and we do not need a sophisticated tool to read the application dialogue, which is a big advantage. The entire application unit is not communicating directly to the Internet because the data passes through the server-side proxy Nginx. This communication middleware is a critical security layer; it can also provide metrics or load balancing features.

Chapter 3

Update Strategy and Redesign

A substantial and significant step on the optimization journey is to ensure up-to-date software background. Keep all parts of the system fresh is a good practice not only from the security perspective. However, we found security as the major benefit because the cumulating number of vulnerabilities in time can be devastating. Every project, even every programming language, is evolving in time to adopt new trends and latest theories. From time to time, it is possible to obtain performance refinement and more system stability only by a software update. If the project becomes inactive or announces the termination, it is better to remove this part from our project or at least substitute its functionality by another one.

During the update process, we can begin with architecture rework. As we mentioned in the previous chapter 2 the application is monolithic, which leads to deploying all parts together, and every developer team works on the same code base. Our vision is to refactor the whole project to microservices gradually. The microservices approach is to split individual units of functionality to separate components. If we were to build the same application using microservices, we would organize the code into several separate components that run in separate processes. Instead of having a single application in charge of everything, we would split it into several different microservices. Describing Tarek Ziadé in his book about microservices [25, page 14]. With consideration of future growth in combination of using this approach we can obtain some benefits:

- Smaller subprojects with narrow usage and more specific purpose

- Better scalability and faster reaction to the short-term payload
- Faster, flexible and easier delivery model

3.1 Software modernization

If we take a look at the table below 3.1, it is clear to see that we are commencing with a four-year-old project. Four years may seem to be quite young but should not be underestimated, as four years in software technology is a long time. Even from the Django project website[1] we can ascertain the major version period is exactly two years, and LTS is three years at most. Also the last support for Python is over, during the days of writing this thesis. Version numbering has some general rules which can help us understand the changes we found in the transition between two versions. The most common version of identification is pretty much the same as describing Python Enhancement Proposals 0440[2] in `X.Y.Z` format where `X` represents a major version number, `Y` represents a minor version number and `Z` a patch number. The leftmost letter represents significant changes in a project which are often incompatible with the previous version. The `Y` changes are bigger than patches and always compatible with all versions under the same `X` versions. And the `Z` changes are usually just small bug fixes or some micro improvements without any interface modification.

Technology	Version	Release Date	Support Date
Python	2.7	July 2010	January 2020 [18]
Django	1.8	April 2015	April 2018
Redis	3.2	May 2016	July 2019 last activity [23]
Huey	1.2	January 2016	February 2019 last activity [14]

Table 3.1: Table shows a version of fundamental dependencies with its initial and ending dates.

[1]https://www.djangoproject.com/download/
[2]https://www.python.org/dev/peps/pep-0440/

3.1.1 Methodology

The manual update process must, in some cases, jump over more than one major version and a lot of minor ones. We want to move from the initial project state, which we will call `old project` in the following lines and target the up-to-date project tag as `new project`. The goal is to find the most effective step-by-step solution to this updated task. The better solution is the less obscure pitfalls we discover during progress. At the theoretical level, we found three most attractive techniques:

1. create `new project` structure beside `old project` and move sources from `old project` to `new project` one by one. During this movement linked dependencies should be updated to latest versions.
2. the `old project` should be updated gradually, following major versions and the source code adjusted to each version
3. take the `old project` and update all dependencies to the latest versions, then adjust source codes to these versions and manage errors

Each project has a different number of dependencies with various ages. The most effective way to keep the project sustainable is to ensure periodical maintenance. From time to time, check the dependencies and make an incremental update if new versions are available. Together with a well-designed set of tests, we can discover possible incompatibilities between two major versions. Tests are also useful to make sure that everything is working after minor updates. If project maintenance ceases at some point then it is harder and harder to perform an update to the latest versions. Because we have so much new modification on each of the dependencies, skipping the major version is usually almost impossible to update incrementally, so in this case we should ignore the first technique on the list.

Our practical `old project` is using almost four years old dependencies, and it also skips some of the major version. The second strategy from the list is less thorough than the first one, but it is also an incremental update. For us, the most challenging strategy is also the most effective. In total, the general update to latest versions will generate many errors, but in cooperation with documentation of changes, we can solve almost every failure caused by the update. The third one also has a significant benefit in that it should not generate so many incompatibilities as the previous two strategies. Because each update can force a particular version of its other dependencies, we must inspect the overall compatibility at every step.

3.2 Design Components

As a part of the update process, we want to rework the project architecture. This project requires better flexibility, increased performance and a more comfortable deployment mechanism. We can meet the requirements by using microservice architecture which is compatible with many other helpful tools because it is currently widely applied in companies worldwide. For this purpose, we need to separate the monolithic project to particular components. The generally accepted concept is a collection of components that are

- Highly maintainable and testable
- Loosely coupled
- Independently deployable
- Organized around business capabilities

The microservice architecture enables the rapid, frequent and reliable delivery of large, complex applications [17]. The transformation of the project to the microservices is an evolutionary process, and a big rewrite of the application is almost impossible [21, page 430]. Probably the best way to carry out this evolution is to create components whenever a new considerable feature should be implemented or reimplemented from the old structure. Each component should be a stand-alone service providing well-described API to communicate with other components in the project, most often by using the standard HTTP protocol. The API we can divide into two types, the first is designed for inner communication only in a private network between components, the second is as a server gateway for communication outside - to the internet. It is a good practice to respect commonly used protocols when devising custom API; it can save undesirable troubles with compatibility in the future.

In our case, described in the second chapter on figure 2.2, the first and most visible component is the main database which we can obtain simply by a separate database engine into service. This component can be self-sufficing as data storage, and connection with it is via standard database protocol using SQL. The second directly distributable component is Redis cache which is short-term storage used primarily by a worker queue. If the Redis storage is dedicated for the worker queue, it could be part of this component. However, other components can use its functionality if we split it to the stand-alone component as well. The worker queue also will become an independent unit. And as the last one, we have the core web server providing computational logic, business logic but also front-end service for an administration environment. In clearly microservice architecture, we should isolate the front-end view from the back-end data part, but in the case of using the Django framework, it is a massive intervention. This step will be part of the next evolutionary step, and for now, the whole server logic stays together in one component.

The monolithic version of our representative application was running in isolated virtual machine mode. For microservice stance is a virtual machine too demanding, every single component should be running its guest operating system with all necessary libraries and standard requirements. The isolation is propitious in this case, and we want to keep using sandboxing. More suitable for microservices is to use containerization.

3.3 Containerization

Containerization is a kind of OS-level virtualization. At first sight, this virtualization can be presented as a shipping container which ensures all tools and environment the application needs. The application itself is a part of this container and provides its services outside for other containers. This communication gateway should be as small as possible to prevent attacks on this application and facilitate better data exchange monitoring.

For the practical part of this thesis, we use container-based framework Docker[3], which is almost a synonym for containerization nowadays. For deeper comprehension, we present in the next line the brief introduction to this framework and used technologies.

[3]https://www.docker.com/

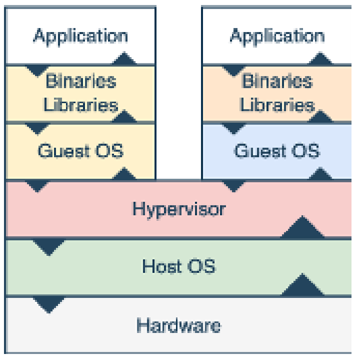
Definition 5. *Image or Docker Image* - Inactive build, including an application with a tools and binaries application, needed to run properly. An Image is created from the specific recipe definition called Dockerfile and always its step-by-step modification of the base image. The application is created when the Image is deployed to the host. For instance, docker provider, cloud, local machine or private server.

```
1  FROM python:3.8 AS base
2
3  # prepare environment
4  ENV APP_DIR /usr/app
5  # create folder structure
6  WORKDIR ${APP_DIR}
7  # copy file to final image
8  COPY Pipfile ${APP_DIR}
9  # open given port
10 EXPOSE 8000
11 # command run after container is created
12 ENTRYPOINT ["python3", "-m", "http.server"]
```

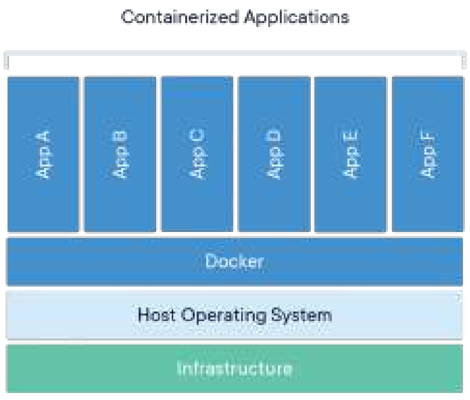
Code Listing 3.1: Example of Dockerfile which is a set of instruction to build Image

In general, containers are more portable not only because of the tight interaction surface but also due to smaller final image size, platform independency and layer system. The layer system is a critical concept similar to VCS. It use caching mechanism to provide a faster build process and to reduce the size the environment consumes. If we take a look at the code fragment 3.1 we can notice each line starts with a keyword which defines the type of action applicable in this step. Every time the line is executed a new read-only layer is created [16] by applying new instruction to the layer before; this gives Docker a capability to cache each layer in the creation process. It means if we build the Image from Dockerfile 3.1, than we change the instruction on line 10 to `EXPOSE 8001`, the build process will inspect the cache before running each line and run the instruction only if it is new. When the engine finds an existing layer from the previous run, it uses a layer from the cache and skips this instruction.

The work with containers is faster and more effective due to the layer concept. This system also saves memory because images can share the layers between themself.



(a) Virtual machine layer structure



(b) Container-based virtualization layer structure [7]

Figure 3.1: In contradistinction to virtual machine virtualization we are able to see the difference the containers are missing Hypervisor and Guest OS layers.

In the next paragraphs we will use the term `scalability` and its two essential classifications in context of application scalability [9].

Definition 6. *Horizontal scalability* - Scale horizontally consists of adding more nodes to the cluster.

Definition 7. *Vertical scalability* - Scale vertically consists of adding resources to the nodes of the cluster, typically involving the addition of processors, memory or disks.

As has been mentioned, the other project run on a virtual machine setup, which is a kind of full virtualized system. This figure 3.1 can help us with the description and comparison of these two approaches. The virtual machine must mediate the hypervisor layer, which is a fundamental environment to host the virtualized operating system with aggregate equipment. User's application is running on the guest os, which has to be furnished with all application dependent libraries, but often it also runs a lot of unnecessary processes. This setup is helpful for a development environment where the developer needs many support tools, but the production environment should be typically equipped only with requisite binaries and application itself. The next part which can manifest itself as a bottleneck of this setup is the hypervisor. For instance, the hypervisor is hosting many machines. At some point, they have some unexpected resources requirements; then hypervisor can overheat and affect the performance of the application operating in the guest operating system. If we want to use scalability to resolve sudden load peak, we are restricted by resources afforded by the host machine.

Against the virtual machine, we have containerization in this case, where the host operating system is running Docker daemon providing an environment for containers. The Docker daemon builds the abstraction layer, which has the same API for containers independent of the host operating system. Each container is an active Image which was built from Dockerfile. The whole stance of Docker on containerization is based on the idea "share what can be shared". Containers share all the parts of the filesystem, which can be shared with the host operating system in read-only [6] mode and each container has its specific mount for writing. In consequence of this sharing, it is possible to run the bulk of containers based on the same operating system with minimal memory space impact and a start-up time of each container is much faster. A full virtualized system usually takes minutes to start, whereas Docker containers take seconds, often even less than a second. If the application is running in the cloud platform, the scale-up is pretty fast, simply by running another application container and distributing a part of the traffic to this instance.

3.4 Cloud Computing

The target condition is to prepare the application to be able to run in the cloud. The idea behind cloud computing is to group a large number of computers, which can provide computation resources, and manage them to provide an environment for hosting software, virtual machines or other services. In cloud computing, we distinguish four fundamental distribution models, whose aim is to provide a different level of abstraction to end-user.

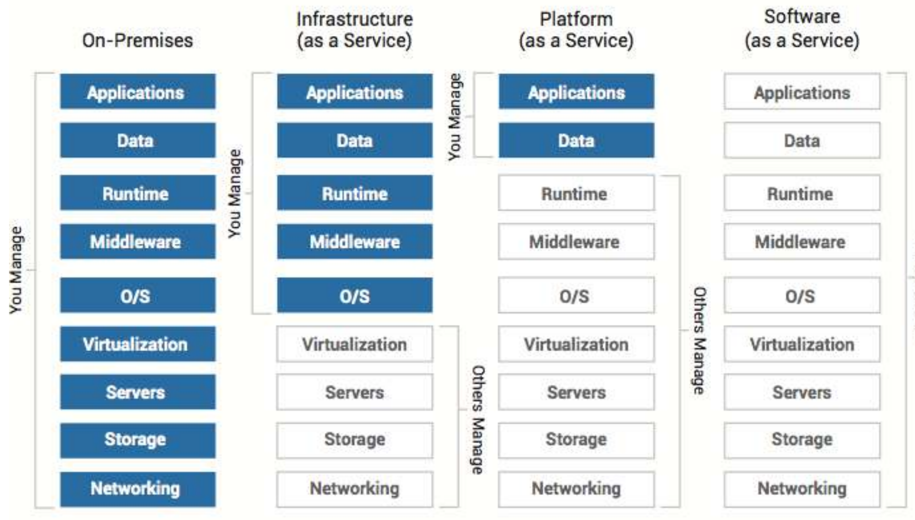


Figure 3.2: Cloud computation models with abstraction layers show particular permission on each layer [2]

Definition 8. *On-Premises* - computing resources, which are provided when required and they are fully controled by the client.

Definition 9. *Infrastructure as a service (IaaS)* - cloud model, which delivers hardware resources such as CPU, disk space or network components. The client controls everything, starting from the OS layer.

Definition 10. *Platform as a Service (PaaS)* - cloud model in which the client receives platform for his application and he has control over the application and data. Everything behind this layers is provider's managment.

Definition 11. *Software as a Service (SaaS)* - cloud model with the lowest level of abstraction. The client works only with the provided software, and he has no control over the technologies behind.

From cloud-based ready architecture, we can achieve another considerable benefit. The container virtualization approach is that containers are designed to provide a new level of abstraction in a cloud. Because if the application is created as a container, it can be hosted on PaaS, which is a second of the four basic levels of distribution models in cloud computing [22]. If we have the application with architecture using LXC we must work with On-Premises or IaaS models and manage the LXC platform for virtual machines. In the container approach, we can run the application in the PaaS model and stay more focused on the application itself without dealing with troubles on the OS level. This approach can also provide new possibilities in composing development team and verified management tooling.

Chapter 4

Automated Delivery

Other requirements on the modern application are flexibility, reliability or adaptability. All these requirements reflect the current information technologies' area situation. Everything in IT has a shorter and shorter lifetime, a bigger concurrency in various sectors because of technologies available to many applications, which should be accessible at least permanently. This process is called Delivery, and it is the entire mechanism on the way from source code to the final application providing its service. Every project which would like to be competitive at some point needs to set up repetitive processes automation to ensure more adaptability. Developers are inventing practices and methods on how to make parts of his job automatic and faster as well, which make repeating processes more efficient; thus, it is a form of optimization. We discuss the basic principles of integration, automation and tools used to implement this principle in praxis.

Definition 12. *Continuous Integration (CI)* - is a development practice where developers frequently integrate source code into a mainline repository, usually several times a day. This process should detect mistakes in code and ensure an automated application build. Often, CI also run automated tests on the prepared build.

Definition 13. *Continuous Deployment (CD)* - approach ensures the development team to produce software in short cycles. Release ability is proved at any point by various prepared tests and use case scenarios. Continuous Delivery also implements instructions to set up each environment and automated deployment.

Definition 14. *Pipeline or CI/CD Pipeline* - a standardized step-by-step sequence of actions connecting the entire delivery process. The pipeline is divided into stages where each stage usually represents logical groups such a source code inspection, build, tests, and deployment. It also defines conditions which must be satisfied to move to the next stage. The stages consist of smaller units called 'jobs'.

The fundamental component of integration is a repository in the version control system, which is storage for source code of the given project. Contemporary VCS provides complex services during the development process, and is not only holding the history of every difference made on the project but also works as documentation storage, issue tracker or even analysis tool. The entire delivery process can be controlled from the VSC dashboard. The origin repository is usually remote server where the team merge their changes and where the pipeline is operating using the last changes version of source code. It is the only repository as the source of code for the application. Every single developer can have his copy of origin repository with custom settings. However, he must interact with it.

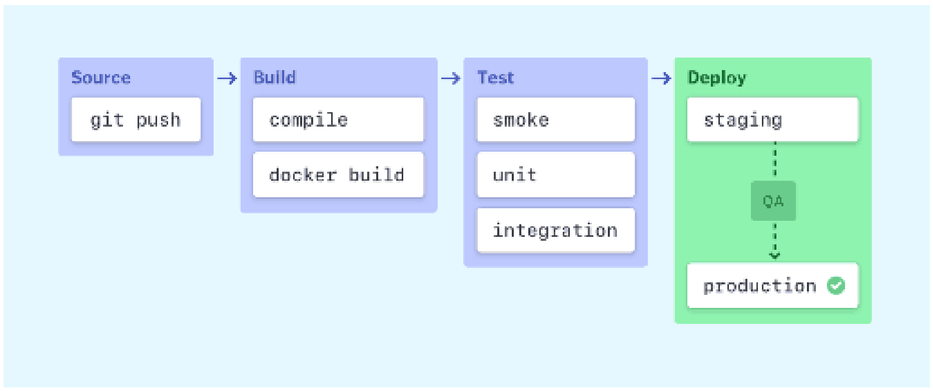


Figure 4.1: Illustration of CI/CD Pipeline clearly separated to the section. Each section includes jobs to be done.

It is crucial to design a specific pipeline which should comply with requirements of the particular application. Every step in the pipeline is optional and entirely depends on the custom design of our application. The pipeline flow is based on dependency, and it means the stage must succeed to unlock the stage after. Dependency is the fundamental property because a well-designed pipeline should prevent deployment when an error occurs. When we are drawing up the pipeline, we are also able to use more control mechanisms such as to run a job after manual approvement, allow run deploy stage only on one branch of the project or protecting a branch from unintended editing.

The key feature of the pipeline is that whenever the source code is changed, it is possible to initiate a reliable and repeatedly consistent build process [4].

As a commonly useful example we can present the figure 4.1 created in the Semaphore framework[1]. We can support four stages, apart from the first stage, which is just a pipeline trigger action, we have build, test and deploy. In the sections below, we take a look closer to what is happening in these sections. The following demonstration and practical examples are using Git VSC, namely software solution GitLab. In the example below, we can see a fragment of GitLab CI script pattern defining stages and job running script.

```yaml
stages:
  - build
  - test
  - deploy
job:
  stage: build
  script:
    - docker build .
  # custom job definition
```

Code Listing 4.1: Example script defining CI/CD Pipeline used by GiaLab

4.1 Build stage

This stage is responsible for collecting dependencies, compiling source code if needed and building an application image in case of using Docker containers. At the successful end of the build stage, we have the latest runnable, or in our case, a Docker image that is deployable to our target environment. When performing static code analysis it is recommended to run before the build stage because it can stop creating an image from code with bugs. However, usually, static code analysis should be done by the programmer.

[1]https://semaphoreci.com/blog/cicd-pipeline

In the practical part, we are using a Docker image as we mentioned in the previous chapter 3.3 to redesign architecture. The main server application is running Python, which does not need compilation because it is an interpreted language, but we must build the application into the Docker container providing a necessary environment. As the base image we are using official Python Docker image with label *slim* which is a mark of minimal version. This is also a part of the optimization from building an artefact perspective because it means the final Image includes only necessary content to run the application and nothing more is unused. For instance, the *slim* image of Python 3.8 is almost five times smaller compared to the full Image, because the full Image contains the extra compiler, documentation or extensive libraries. As a consequence, management and transfer of the final Image is much more comfortable.

If we have a build stage of the pipeline prepared, we must run it to be complete. For this purpose, GitLab is using a concept named GitLab Runner and, in general, we will use *runner* designation. The runner is a service hosted on the server machine which has sufficient resources and is told to run pipeline jobs. Depending on support services architecture, GitLab Runner can be hosted on a production server or another server set for running the pipeline. However, GitLab Runner is the only pipeline executor, and it is responsible for every stage, even for delivery to the production environment. The runner is designed as a microservice, so it is stateless, and in case of using more runners, it is possible to run each stage separately on a different runner, but the dependencies must always be satisfied. The process of each step is carefully recorded, and after the stage is terminated, the runner reports if something has failed.

4.2 Test stage

In the test stage, the successfully built Image is tested. This is the part where we inspect if the new Image meets all requirements designed in tests and if the application starts without complication. This stage is optional, but it is highly recommended to integrate into a pipeline because it can automatically discover many failures at the time. It is also desirable to focus on the test report in case of negligence, because elaborate information can help the developer to detect the problem and fix it quickly. In contrast to the successful test, where the crucial information is successful and probably a green colour for easier eye detection.

A well-designed set of tests can save a lot of time and avert deployment of the flawed application. We have many sorts of tests available, so we will briefly go through some types suitable for pipeline testing.

4.2.1 Unit test

Unit tests are usually the smallest tests [25], which are responsible for testing particular functions on class level. This type of test is specific due to its isolation because functions are tested with no application context [17], so we need to prepare the whole context artificially. This can be useful when we want to simulate different kinds of situation, but it can also set up an unreal environment and thus miss the test of core functionality. Unit tests are created to verify the functional correctness easily and rapidly. Functions often expect some states as input. Therefore unit tests are also responsible for preparing the environment for testing. Apart from testing standard functionality, unit tests should be well designed to scrutinize functional behavior in edge cases. We hope the well-prepared set of unit tests is the point where we catch most of our bugs [17].

4.2.2 Integration tests

These tests are started to hold up an entire system [4]. They check application configuration; they also check whether an application can connect other subsystems and services. Often integration tests are responsible for the internal connection, instead of external sources or inaccessible sources can be replaced by mock. The mock system is a dummy replication of a real resource, which provides exactly the same communication, but customarily serve no data sources. The mock is also a good practice to make tests more repeatable without risk of an external source overheating.

Integration tests also check that our service is ready to provide its functionality as expected [1].

4.2.3 Service tests

Service tests are one of the latest tests in the pipeline which are focused on a more complex service functionality. We can test various scenarios to be sure the application works properly. They should be strictly directed against the service core functionality. In general, these tests cover a bigger scope, so when they failed, it is harder to detect what is broken [17]. Tests are complex, therefore running all designed cases usually takes more time than unit tests.

4.3 Deployment stage

As the last step in the pipeline, we must have application deployment. In this stage, we have successfully created an application image which was tested correctly by the previous stage and which is now ready to be published. There are customarily more deployment environments like testing, instability, or production. We obtain more flexibility because of using containers for hosting an application which can have a various hosting platform. The container deployment model can be compared to a shipping container which is a universal transportation box. If the means of transport has a container platform, it can transport this shipping container. The same situation is with Docker container deployment; we can deploy this container to whatever platform which is set up to host a Docker container. For example, a cloud, virtual machine, or a bare-metal server as well.

The important feature of the deployment process is the trustworthy and consistent replacement of previous application version. Moreover, the replacement process should be as fast as possible and with a short connection loss or without connection loss at best. These requirements should be well designed to satisfy application needs, or it can be solved using some additional management service, for instance, Kubernetes, Mid Vision, Docker Swarm or other helpful orchestration tools.

Consider using an orchestration tool, which can help us with other useful analysis during application runtime. Good practice is to observe the application's health or functionality to be able faster detect troubles, but also to carry out some essential application monitoring, log collecting and performance information analysis. All these things can be gain by using additional tools. In our case, we are using simple application health check and log collecting, this functionality can be resolved without orchestration which can be sometimes difficult to operate.

Chapter 5

Optimization Outline

In this chapter, we will focus on the optimization of the application itself. We have an application based on the latest technologies, using an automated delivery system which is flexible and easily scalable. Thanks to this setup, it is much easier to resolve temporary load growth by scaling up or adding resources. However, from a long-term perspective, it turns out that a better approach is to clean and optimize the core functionality, which leads us to keep sustainable codebase and a stable application. Typically we want to discover application bottlenecks, to understand its issue. Then we have two options: optimize the existing solution as it is, or try to create a new conception which resolves the problem more effectively. If it is possible during the development process, the good practice is to refactor wider code surroundings when we are implementing a new feature.

Many books and articles are direct when presenting optimization. The representative example we can read in the Python Cookbook: While the first rule of optimization might be to "not do it," the second rule is almost certainly "don't optimize the unimportant." [3] We should be really careful when we want to focus on optimization, as the essential part is to profile and deeply understand the application. Also an application earmarked for optimization should be verified as fully functional for the purpose for which it was created. We should have available a full set of well-prepared tests, which can help us confirm that application work correctly after the changes are applied. After we have the support tools prepared, we can focus on optimization itself.

Definition 15. *Bottleneck* - part of the system, which is fully loaded for most of the execution time and which cannot be carried out more naturally. This section does not allow the whole system to speed up, although it can have enough resources.

Generally speaking, optimization Python application on a programming language level stands on the choice of correct data structure or suitable algorithm. The built-in data structures in Python are written in C programming language, due to elementary operations using core data structure that are always faster, or at worst as fast as using the best Python designed structure [3]. So one of our objects of interest will be proper usage of algorithms and data structures. In the case of the algorithm, the characteristic illustration is recognition of small variances in using a sort algorithm. The crux is to think about the point of the problem we are solving. How big is a dataset we are processing? What is the structure and which fragment of the content is our part of interest? Does the data change during our process? Imagine the situation when we need to sort a sizeable invariable set of comparable objects by a unique identification number, then we use an advisable algorithm which can effectively sort a significant amount of objects. And the opposite situation when we need to repeatedly insert new objects to a data structure which is already sorted, then it is better to use a different algorithm, probably Insert sort.

Sometimes in the discussion about optimization, programmers argue: "I do not want to lose time thinking optimization over, I just simply add more resources.", which can be the solution for sudden performance augmentation. Thinking over more profound optimization at the very beginning of the project can also be counter-productive because in this phase is still a pretty unstable project and excessive optimization can make it worse [13]. But when the project reaches a more stable phase, it is highly recommended to focus on code fragments' analysis and resolving bottlenecks. This is a good practice not only from a long-term perspective, but it can also help the application with a higher load and to reach high performance. Another aspect is readability, optimization refactoring should not make the code unreadable, these changes may ensure great performance betterment, but when the code is understandable others cannot work on it.

In the following sections, we want to discuss the specific situation discovered in the course of work on the practical part of this thesis.

5.1 Bottleneck detection

Identification of parts which slow down our application and analysis of runtime is essential when we want to speed up. In most cases, this slowdown is caused by relatively small fragments of code which we need to discover. Closely related to performance analysis, we can see in the books another term - Hot spot.

Definition 16. *Hot spot* - region of a computer program where a high portion of computational time is spent during execution.

If we want to detect some bottleneck, regardless of suspicion where to search for trouble, a useful tool is profiling. Profiling is a technique that allows capturing parts of an application runtime. Monitoring is done by using a profiler, which is an entry point to the application when pursuing a recording. During runtime, the profiler is recording execution time taken by function doing its job and recursively execution time of every named inner function. Apart from the recording time, the profiler is capturing various other useful pieces of information. For instance, memory consumption, CPU time per function, number of function calls, et cetera. The profiler output is usually a standardized measurement `prof` file, which is binary readable by numbers of analysis tools and contains information typically as a structured tree graph. The result can be interpreted as a text table with line by line values, which can require more experience to read and understand this output. However, the result can be represented as a graph visualization, which is helpful when we want to obtain a general overview of what is happening in the entire application.

To get a general look onto the application we can use profiler at the topmost function, but we can expect the tree graph to be spread out and confused. In this case, the visualization using colours in the graph can help us to detect overheated parts and focus on these fragments in the next profiler run. In figure 5.1 we can see tree fraction with an overheated function where red colour signals express slowest nodes, the output from Silk framework[1].

Profiling is beneficial, and it also works well in combination with other tools. When we are inspecting application performance, we also use benchmarking. Benchmarks are tools,

[1]https://github.com/jazzband/django-silk

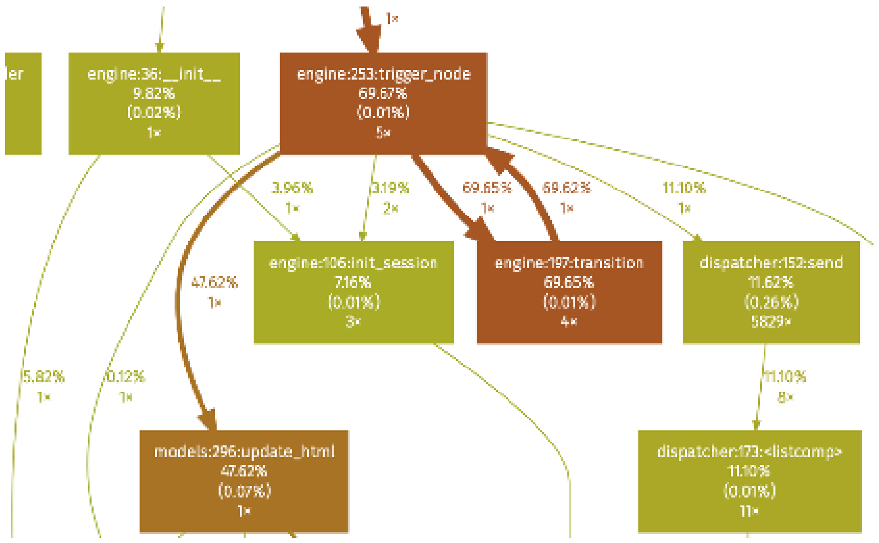


Figure 5.1: Graph displays a part of the profiling call tree. A helpful tool which can be used to discover critical application sections. In the node we can see the name and location of the function, amount of computational time spent in this function and number of function calls. Color indicates the time taken by this function where red color stands for more time.

and scripts are used to assess the relative performance of an application or an object by running a number of tests against it.

5.2 Design level

Thinking over the design and intent of application parts can help greatly with optimization itself, therefore before we come up to optimization on the source code level. We should focus on the effectivity of disputable parts. Little changes in architectural design can have a huge impact on performance. For this purpose, we again need to look deeper into how the addiction treatment program is designed and how the specific sections should act. We have an interactive therapist who should substitute for a real therapist and who can help smokers to quit, as we present in chapter 2.

In our case, the most inefficient part is the task queue which is liable for data preparation each day. At the start of the day, the queue has a set of scheduled tasks, which the queue should go through and distribute to all notifications. The notification is a tool, which helps to keep interaction with the user via dialogue, and it is a way to support him during the smoking cessation progress. After the user accomplishes his daily session, the next session is automatically prepared for the next morning. When the user is active in the addiction treatment program, he should receive an email or notification which reminds him to go through the particular daypart [5]. Scheduled tasks are essential for the application to work properly, because they are responsible for data preparation and they also should be carried out as fast as possible.

Once the user enters the program, for the most effectivity he should be active every day, otherwise the system needs a scenario on how to get the user to return. If the user stays inactive, after a few days the addiction treatment program becomes meaningless and the impact of this is due to program suspension. Then, after 6 - 7 days, the user can be marked as inactive. When the user becomes inactive, he must start the program again from the very beginning.

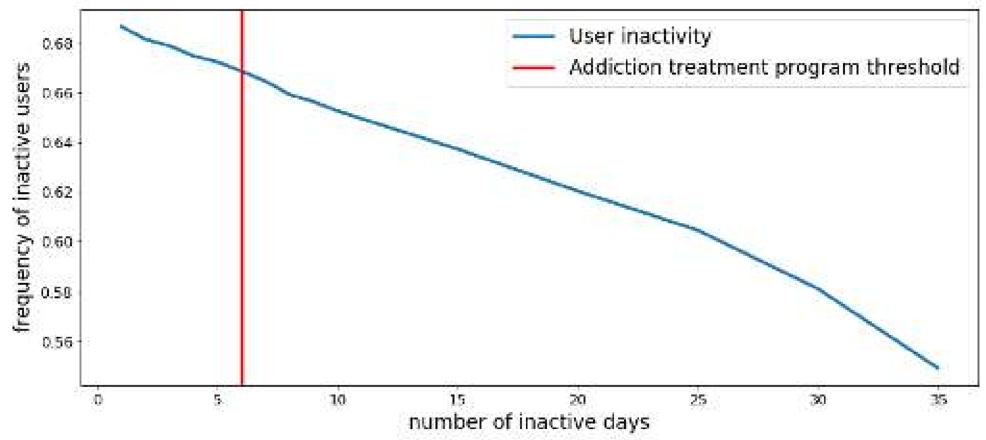


Figure 5.2: Graph visualization of average user inactivity. From the end of the first day up to more than one month. Threshold represents the virtual boundary after, which the program must be reinstated if needed.

In the figure above, we can find that the immediate departure, which starts on one skipped day, is more than 68 % and rises to values around 58 % of inactivity for 30 days or more. This behavior supports the fact that quitting smoking is hard to manage, but also it can point out a large number of users who want only one glance at the application and never return back. Most people probably use the application in a trial mode and a minority update afterwards to the full version.

From the optimization perspective, we can see huge potential. One of the fundamental principles in the optimization of almost everything from the core system, algorithm, database access to pure computation, is to not spend time with something which is not necessary. We are using this principle in CI/CD, when we are creating the least possible container to host the application, and it is essential also in optimization. We can precompute data for all the users who are active or inactive for six days and less when the user exceeds the threshold, then he is removed from the queue. This optimization helps us to focus on active users and to prepare the data much faster because computation is processing approximately 40 % of all registered users. If the user wants to return back to the quitting smoking program, he must commence again from the very beginning.

5.2.1 Decision Tree

The Decision Tree algorithm is the core model and functionality of this application, which due to the framework is designed for a universal logical-based application simulation. It is excellent and helpful when the application is in the development phase, but when the working model is done, the complexity of a framework can be a burden. Some of the parts can be recreated to be more specific in sections where it is possible. For instance, the framework is using its own expression parser, described in this chapter 5.3.2, which is more robust in processing custom expression, it also allows some kind of simplification. Overall, the parser is not so much different from the built-in Python expression evaluator, but the biggest difference is that the built-in evaluator is written in C compared to a custom solution. In consequence, if we are able to substitute the parser, we can gain on average an almost 100 times faster evaluation. The system admin can provide a simple tool for expression correctness verification.

Another appreciable optimization concerning the core model is size reduction. From the early phase of the project, the internal model has many branches and nodes, which seems to be unused in the production application. It could be the facilitation of the computation process for the core algorithm because it must go through the whole graph, including inactive parts, which is time-consuming. The detection of these parts is dependent on the requirements of the addiction treatment program and it should be detected in cooperation with designers of this program.

5.2.2 Database model

An auspicious impact on effectivity can provide some modification related to the database and storage model. Various benefits can ensure an adjustment of column types in the database table, for example, when we work and store data in JSON form. Opensource database PostgreSQL support JSON data types since version 9.2[2]. The support also includes more complex operations with JSON and even processing data as jsonb.

The jsonb is a binary representation of JSON object, which can be significantly faster than plain text processing, and it also supports indexing. It can be beneficial in the optimization process and definitely in a situation when we have an old project, which can obtain this JSON support only by a plain database update.

5.3 Source code level

The optimization of source code itself is more about programming and experience with a given programming language. On this level, we target data structures and algorithms used in critical fragments of our application. It is important to understand the data flow inside functions and comprehend all properties of used data structures in the scope of the inspected code fragment to be successful in optimization refactoring. Sometimes the inefficiency can originate only from misunderstanding during implementation or directly from a programmer's mistake. We want to discuss some techniques which are universal in Python application optimization and which were used in the practical part of this thesis. Comparison results of using optimization strategies will be described further in the next chapter 6.

Changes we are presenting in this section should not affect application functionality; they are more focused on the implementation itself. Sometimes the optimization can be obtained only by updating core frameworks to the latest versions.

[2]https://www.postgresqltutorial.com/postgresql-json/

5.3.1 Data structures

An important part is using different data structures in different situations and the ability to find the most suitable one. In the code example 5.1 we can see a simplified code fragment, which is a common cause of wrong data structure choice [3] [15].

```
1 class Parser:
2     def __init__(self):
3         self.names = [v["name"] for v in variables]
4
5     def parse_name(name: str):
6         if name in self.names:
7             # Do something
```

Code Listing 5.1: Example of bad data structure selection

The class has an attribute of `names`, which stores a list, and the only usage of this attribute is with `in` operation on line 6. Interpretation of this code is that we create some data structure then we ask if a name we want to process is presented in this data structure. The substantive decision is in data structure selection. List in Python world is Linked list data structure [24], so it is good for operations with a head or tail, but if we want to accomplish this `in` operation, then the time complexity is linear $\mathcal{O}(n)$[3]. The List must go entirely through its dataset and ask each element if it is equal, which is directly proportional with the number of elements in the List, thus the time complexity can significantly grow.

If we want to perform the only `in` operation, we can also use the Set data structure. The Set is internally implemented as a hash table, where keys are elements of the Set, and values are always null. A huge benefit of using Set is when we want to do many natural set operations like intersection, union or subset. In this example, we use a subset operation, whose time complexity in the case of Set is a constant $\mathcal{O}(1)$. In contrast to List implementation, the Set is a thousand times faster and even independent on the number of elements. In this case, the replacement of List by Set data structure has no influence on the functionality, but

[3]https://wiki.python.org/moin/TimeComplexity

it can significantly increase the performance, especially when we assume a large number of elements and many repetitions of a function call.

5.3.2 Instances

Appreciable optimization can also be attained by reducing the number of unnecessary instances. To assume a simplified example represented by code fragment 5.2, where we can see class `Engine`, which is a part of core functionality, and its task then is to process a decision tree algorithm. The model graph has thousands of decision edges, where the algorithm must parse and evaluate some predefined expression. Some of the expressions are processed multiple times because the model allows backtracking. Every new instance of `Parser` class is creating Backus–Naur (BNF)[4] parser, which is a notation model of context-free grammar[5]. Creating this instance is not trivial and creating hundreds or thousands of instances of Parser class is quite expensive from a computational time perspective.

```
class Engine:
    def function_one():
        parser = Parser(self.user)
        result = parser.parse(expression)
        # do work

    def function_two():
        parser = Parser(self.user)
        result = parser.parse(expression)
        # do other work
```

Code Listing 5.2: Example of unnecessary creating many class instances

The acceleration can be done by moving `Parser` initialization to `Engine` constructor and to save an instant reference as a class argument. Then we can use parser repeatedly with refreshing inner structures between parse expression calls. In implementing this reduction, we

[4]https://en.wikipedia.org/wiki/BackusNaur_form
[5]https://en.wikipedia.org/wiki/Contextfree_grammar

have an excellent chance to save computation time without changing the logical application flow.

5.3.3 Generators

The generator is a pretty powerful mechanism, which can provide effective optimization of cycles. The generator can be a pleasant way to replace usual iteration through List, dictionary or other Python structures that you can traverse through all values. It holds information about how the data source is created, but the data are not evaluated until we do not need them, the concept behind the generator is practical usage of Lazy evaluation [20]. This principle in Python can help work with almost infinite structures without considerable memory consumption and also final computation is significantly faster, because the intermediate results are not computed if not needed.

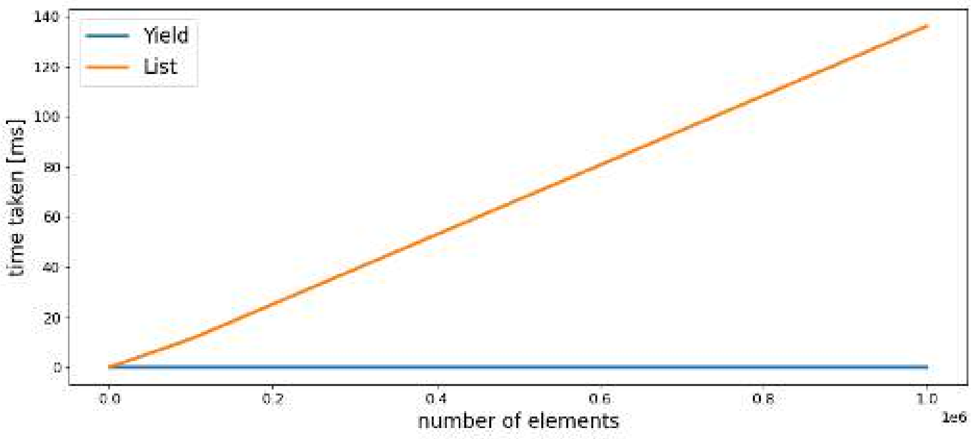


Figure 5.3: Computation time compare list and generator iteration. The iterator takes constant time on average, which is shown by object initialization. From the opposite side, the list approach takes time directly proportional to the number of elements.

In picture 5.3, we can see that in using generator representation only initialization is consumed and computation is executed only when the data is requested. Whereas List as a core structure can be time-consuming, especially when processing a large number of elements due to it being a strict evaluation, which must be carried out immediately. Moreover, the temporary results should be stored, and it costs memory, which is growing with a higher number of elements, and it can even exceed limits. Common cases should not produce so many problems, but when we have many consecutive cycles, and we are interested only in the final result, the generators can ensure huge mitigation.

5.3.4 Slots

```python
class Example1:
    __slots__ = ["year", "month", "day"]
    def __init__(self, y, m, d):
        self.year = y
        self.month = m
        self.day = d
```

Code Listing 5.3: Data storage class using slots optimization

Another method that achieves a better application performance is to use Python slots. The impact of this method is more on memory consumption and management, rather than speed. The idea behind slots is pretty simple: by default the Python interpreter is not restrictive and it works without knowledge class arguments, so the internal storage functions as a dynamically allocated dictionary. Until we use `__slots__` internal class variable, which hold the class argument names and gives an interpreter more information about the fixed number of arguments, then the interpreter can be more effective in memory management [3]. Proper use of slots can help to curtail the amount of allocated memory up to tens of percent[6].

For the most part, slots should only be used on classes that are going to serve as frequently used data structures. Usage of `slots` brings about some additional behavior. A common misperception is that `slots` is a tool to prevent the user from adding new attributes to the class [3]. Although, this is a side effect of using `slots`, which are originally designed as an optimization tool.

[6]https://book.pythontips.com/en/latest/__slots__magic.html

Chapter 6

Experiments and Measurement

We described optimization techniques, and this chapter will be focused on practical implementation, improvement measurement and presenting obtained results. The optimization process is a more complex process than straight utilization of presented methods and recommendations. At first, we should define, usually with the project manager, what the goal is that we want to achieve. Generally, the objects of interest are closely related to resource and can be divided into several main classes:

- increase application speed - response time, computation time, user feeling, et cetera.
- reduce memory consumption - RAM, disk space, databases
- cut down CPU time
- reduce network communication - requests quantity, data exchange

Almost every time the request on enhancement in one of these classes results in bigger requirements than any other class. For example, if we want to improve network communication by reducing the number of transferred bytes, we can use compression before sending the data, which will cost more CPU time consumed by the compression algorithm. This rule can have some exceptions in situations when the application in some parts is focused on the computation of unnecessary things. We intend to speed up the application core computations and focus on sections, which can be redundant. The assignment also allows the modifications, which can cost slightly more resources.

6.1 Test environment

All measurements and tests we made on MacBook Pro, 2016, 2 GHz Dual-Core Intel Core i5, RAM 16 GB using latest available technologies as present chapter 3, specifically Python 3.8.1, Django 3.0.6, Docker Engine 19.03.8.

6.2 Computations

As we mentioned, in the practical part, we are focused on speeding up internal computation processes. The measurement criterion to evaluate the applied adjustment is effective and for determination of quantifiable improvement we use computational time and closely related `speedup` ratio.

Definition 17. *Speedup* - is a ratio which tells us how much faster a task will run using the application with the enhancement, as opposed to the original application [10].

For our purpose we use overall speedup, which use general computational formula:

$$Speedup = \frac{Execution\ time_{old}}{Execution\ time_{new}} \tag{6.1}$$

One of the biggest improvements we obtain by ignoring inactive users, which we present in previous section 5.2. Implementing a simple threshold we reduce up to 64 % of unnecessary precomputations (figure 5.2), almost for free. For users who want to restore the program, they are able to restart the process from the beginning. This enhancement is not included in the overall results and graphs, because it is not accurately measurable due to a nearly unpredictable amount of work for each day of computation. From the several measurements of two different implementations, we gain the average speedup `4.3`, which is significant acceleration.

Another optimization is performed on the source code level by applying different methods and practices. In the following text, we use terms for an individual class of optimization, where every class arises from the previous version by using a successful new technique. The success, in this case, is a technique which is correctly implemented, which has an acceptable impact on application conduct and which helps to speed up computation. This evolution starts from class H0, in which is the application without optimization from our point of view.

- H_0 - application without optimization
- H_1 - add slots in data holding classes
- H_2 - optimize expression parser and evaluator
- H_3 - reduction number of new `parser` instances per request
- H_4 - optimize interaction with database
- H_5 - optimize core engine internal iterations

In a more complex software project, it is difficult to discover code fragments, which can block the rest of well-performed sections. Therefore we need some aid for bottleneck detection and general analysis. We use various tools such as to request senders, time measurement scripts or multiple profilers. The fundamental tool is a profiler, introduced in section 5.1, whose output understanding can be truly helpful for software fragments analysis. Profiler measurement is focused on time measurement, and the report includes separated data for each function about a number of calls, time spent in the function and all subfunctions or time spent in function itself [8].

In the example 6.1 below we can see the profiler output of measurement application with no source code level optimization. This output was produced by computation fragment for test user on Session 4 and results are filtered to display mainly our editable source code, which is most relevant for optimization instead of third-party libraries.

```
ncalls   tottime   percall   cumtime   percall  filename:lineno(function)
1        0.000     0.000     1.100     1.100  serafin/content/views.py:22(get_session)
1        0.000     0.000     0.872     0.872  serafin/content/views.py:54(get_page)
1        0.000     0.000     0.807     0.807  serafin/system/engine.py:519(run)
2/1      0.000     0.000     0.807     0.807  serafin/system/engine.py:191(transition)
13       0.001     0.000     0.698     0.054  serafin/system/expressions.py:125(__init__)
22       0.631     0.029     0.631     0.029 {method 'execute' of 'psycopg2' objects}
2        0.001     0.001     0.302     0.151  serafin/system/engine.py:172(traverse)
12       0.000     0.000     0.097     0.008  serafin/system/expressions.py:290(parse)
```

Code Listing 6.1: Profiler output example of computations on Session 4

If we look closely at this particular output, we can see ordinary functions behavior, but then the execution reaches the custom expression evaluation phase. We can notice that for each expression the algorithm must evaluate and trigger a new initialization of a `parser` object, which always launches the same database lookup. After a deeper understanding of how this algorithm was designed, we realize that this recurring resource allocation is superfluous and expensive in case of CPU time. By implementing simple changes to this class, we can achieve the reusable expression parser, which is a part of an `engine` object and can be accessed when needed. Therefore we curtail the number of dedicated `parser` instances as well as database communication, furthermore the `engine` object becomes even more robust and consistent. In the edge case, the previous core `engine` could be inconsistent because it loaded the whole set of system variables for each expression evaluation. If some system variable was changed between the evaluation of two expressions, the result of the same expression could be different. This optimization is in text marked as H_3, and from the final measurement, this modification turns out as the most effective.

Another practically used method, in general, is to look at functions with considerable time per call. If we want to save computational time, we must be sure that the algorithms do mainly what should be done, without any significantly needless operations. Customarily the database access can be a possible hot spot for applications, especially when the communication is realized over the network, so it brings some performance improvement to make a strictly necessary exchange in this part. In our user case, it is useful first to bring the user from the database and save them again after the computational part is done. Additionally, in this computational part, the core algorithm uses only one data variable, which is part of the user object; therefore, we can save only specific fields instead of whole objects[19]. In the text this improvement is tagged as H_4.

6.3 Data operations

Other implemented optimizations are more about operations with the data in core modules. Proper work can help to save memory or time.

As a first optimization, we add `__slots__` to classes, which are data holders, and the server creates many instances of these classes in runtime. This optimization is marked as H_1 in measurement. By defining this variable in a class, we are telling Python interpreter to use the different internal form to store arguments [20]. This adjustment should help to save memory consumed by initialization, and theoretically, it can slightly speed up access to data. In our case, in combination with other techniques, it helps to partially improve the computation speed and approximately 20 % of memory.

Mainly in expression `parser` there is a big potential in proper data structure selection. Crucial to this case is to know what kind of data we need to store and exactly which operations we do with them because different structures require various types of manipulation. If the whole manipulation set is a combination of operational characteristics for sundry data structures, then we can decide to use multiple data structures each with different intent. On the other hand, we can choose a data structure, whose operations are more frequented. An appropriate example for time spent by a particular operation can be a membership test.

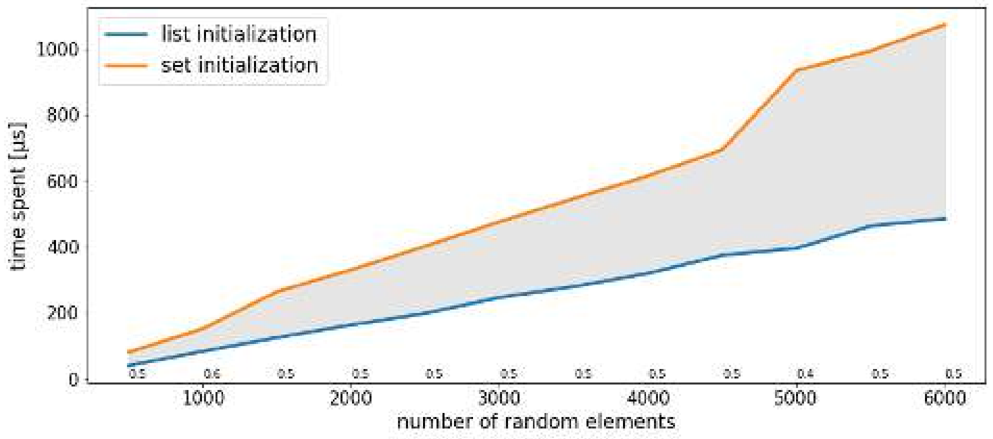


Figure 6.1: Comparison of two different data structures - List and Set. In the figure we see an average initialization for both data structures with a variable number of elements.

In figure 6.1 above, we can see a comparison of two data structures as possible choices for usage with a membership test. The model collection has 2000 various elements and it was created as a representative situation from the application environment. The measurement was accomplished on data structures with different amounts of elements for a deeper view and better understanding of the surrounding context. We can notice that initialization of booth data structures is directly proportional to the number of elements and List is approximately two times faster than Set. This behavior is expectable because Set must perform hashing on a new element.

From initialization, the situation is not so clear to see, the biggest advantage of Set is in membership test on this data structure. On the picture 6.2 we can see the practical measurement of the same function on the different data structures where we can notice two different time complexities $\mathcal{O}(1)$ in case of Set and $\mathcal{O}(n)$ in case of List.

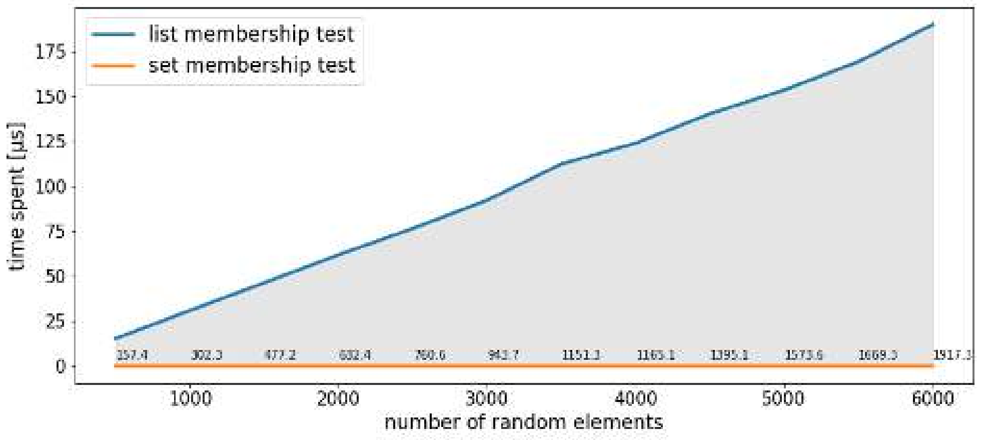


Figure 6.2: Comparison of two different data structures - List and Set. The figure shows time spent by "in" operation with changing number of elements.

In total, it can save valuable seconds, when this principle is used multiple times during algorithm runtime in several different places. For the future, it can also have the potential to save more computational time if the number of active users increases growth. This optimization is marked as H_2 in the text.

The last implemented optimization H_5 is focused on the reduction of redundant iterations.

During the development process it is natural to store pre-computed parts in temporary variables. This can definitely be useful in the debug process when we need to repair some sections and it is necessary to inspect data flow. However, after some time and various sets of tests, when the functionality is verified as working and without bugs, we can analyze the source code and make some non-destructive adjustments, which can help to improve application runtime. One of these adjustments, which was used in the practical part, we can find in a code fragment 6.2. If we discover variable `edges` as temporary, we can reduce the number of loops in code by quite simple modification. In the first example, we can see three loops doing the job, which can be done by one loop and two if they are statements. This modification is non-destructive and it can help to speed up the algorithm runtime, especially in a case where we have a huge number of edges to process.

```
# example 1
edges = get_edges(edge_storage) # loop
special_edges = get_special_edges(edges) # loop
normal_edges = get_normal_edges(edges) # loop

# example 2
special_edges, normal_edges = get_edges2(edges) # loop
```

Code Listing 6.2: Examples of two implementation presenting internal loop reduction.

6.4 Results

All these changes individually have the potential to improve the application's performance. For test purposes, we pick a representative set of `Sessions` which are the form of how to define job instructions. In this case, the `Session` must be pre-computed for the test user, and we are interested in the duration of computation runtime of the application on different levels of optimization as defined above. The intention in collecting the set of `Sessions` was to test applications in distinct situations and spread over a large number of cases.

In the table 6.1 we can see computational times in milliseconds for each pair `Session` and the application's optimization level. At a glance, we can see the considerable improvement between H_2 and H_3. It is also discernable that each optimization helps to improve a different kind of Session and modifications together achieve the performance improvement up to four times, in some cases.

The following graph 6.3 is visual representation of measurement.

Time [ms]	S_{139}	S_{140}	S_{141}	S_{142}	S_{143}	S_{144}	S_{145}
H_0	1236	1219	1204	1114	1338	1236	1183
H_1	1030	1079	1149	1134	1008	1153	1055
H_2	1036	1081	1048	1132	1032	986	1058
H_3	368	491	517	501	461	494	600
H_4	377	411	398	378	431	382	394
H_5	353	377	360	357	375	357	388

Table 6.1: Measurement of computational times on selected sessions after apply optimization techniques. Seven representative sessions can be found in head line. In left most column are optimization levels, each new level is created from previous by adding new optimization.

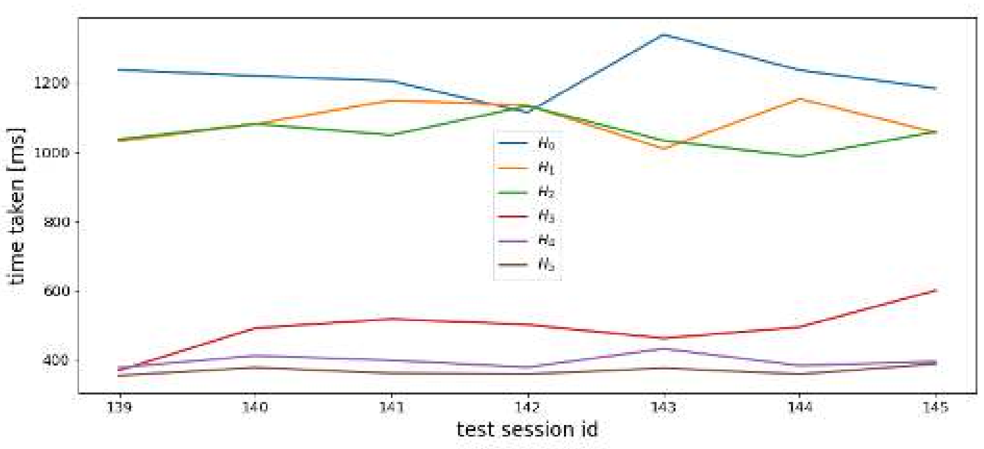


Figure 6.3: Computation time in a task queue for a representative set of Sessions. Visualization of data from previous table.

The interesting point is to see the computation time for the `Session 142`, which was almost the same after the first two optimizations, but the H_3 significantly reduces the number of `Parser` instances and this was the biggest improvement. Interpreting the graph, we can also notice the evident impact of the database access optimization represented by H_4 line.

To end, we should present the second graph 6.4, which shows the cumulative speedup after applying each optimization technique. From the collected data we find that the final version of application causes threefold speedup through all representative `Sessions`.

It is almost impossible to measure the overall impact of these modifications on the whole

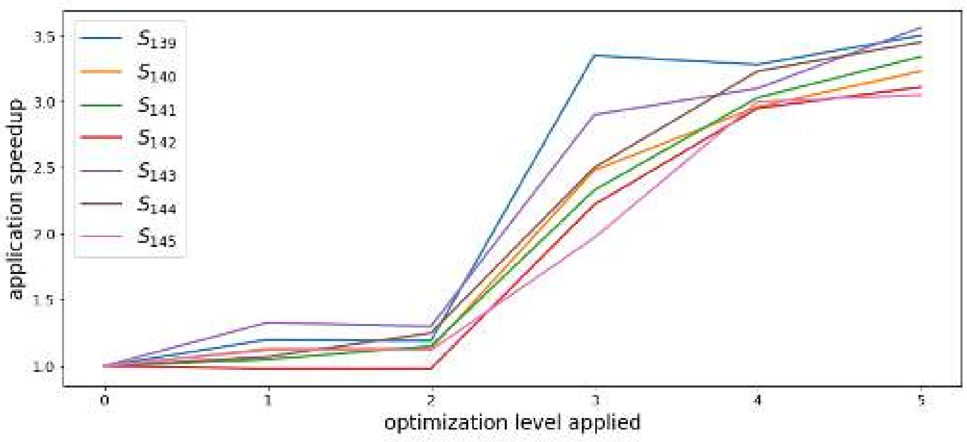


Figure 6.4: Speedup visualization of computation process per Session

application because the amount of work for test users is complex and different each day. Theoretically, we have an option to revert all data after computation, but it is too complicated, and it can cause inconsistency. But the total runtime results are measurable in minutes with 1-2 minutes deviation.

	No optimization	**Inactive Users**	**Source code**
computation time [minutes]	56	13	4

Table 6.2: Representative task queue computation time after major optimization steps.

In the table 6.2, we can see the overall results of runtime on the representative task queue. The practical modification of an existing application can reduce computation time from 56 minutes to approximately 4 minutes in the final optimized application. To better understand how large the amount of time was, obtained by ignoring inactive users, we present the indicative computation time on a no-optimized application with a reduced number of users.

6.5 Data transfer

The last section is about data transfer and the additional potential for optimization. When the server-side prepares the data for a user, it must send pages to the frontend, e.g. a mobile application or web browser. In our case, this communication is solved by standard HTTP communication using JSON as a data object. In general a good practice in communication between two services maintained by the same team or provider is to keep the data exchange as small as possible. Because it can improve data processing, reduce the size of transferred information, or simply improve readability. All in all, clean data exchange from redundant content can bring huge benefits.

Page name	length	size	length reduced	size reduced	gzip compression
`EE01.1`	4863	5.5 kB	1616	1.6 kB	797B
`E01.5a`	25751	32 kB	6083	5.9 kB	2.5 kB
`EE01.2`	18970	23 kB	4153	4.1 kB	1.9 kB

Table 6.3: Data transfer reduction on selected Pages. First we have original size, then size after removed redundant content and finally additional compression.

In the table 6.3, we can find example pages with present data size compared to possible size reduction gained only by removing surplus information. This effect can be supported by using additional gzip compression, which can reduce the final size even more. Experiments with this optimization show the opportunity to curtail the effective `Page` size up to more than 12 times in case of `Page E01.5a`, as we can see in the table.

This improvement can also have an impact for the client because loading `Pages` can imply an impression of speedup. For this thesis, we can refer this optimization to future work because it requires some modification on the client-side, which is not part of our scope.

Chapter 7

Summary

The overall target of this thesis was to modernize an existing web application, which in some parts uses obsolete technologies. Technology update is crucial in software engineering because it can ensure a more secure code, and every new version should fix other bugs, either by major or minor cleaning. It was a great experience to successfully solve the modernization of a complex project with many dependencies and links to other third party sources.

The application was also traditionally created as a monolithic software project, and we made the first steps to recreate it as microservices, which can produce some new possibilities in the development and simplify the maintenance. The fundamental advantage of this approach is automation deployment closely associated with containerization using Docker framework. Automation is the essential equipment of a modern project because it can help save time on repeated actions that the project requires, and in our case, it can also help us with a deployment production version to AWS. To set up this environment properly for the requirements of this project was a challenging task, but now it can be reused and enhanced in future development.

Last but not least, the important part was the optimization of the application itself. Part of this task was to accomplish the analysis of a core framework and understand the whole concept mainly from source code without detailed documentation. Afterwards, the challenge was to select relevant possibilities and try to improve them based on previous analysis. This was a great verification of software analytic skills and gaining new experiences.

7.1 Conclusion

The main success was definitely an improvement in the application. The expectation for this task was to gain at least some seconds from the previous version, but after implementation, the result was about counting minutes in our favor. This was possible because we can focus on a particular domain of use and modify the most general parts, which often leads to cutting off unused states and to be more focused on the main object. Consequently, the optimization can help an application with local overload and with preparation for new users.

In the deployment part, it was important to prepare an application for the new platform and integration. The base for this purpose is configured and works well. The whole setup is ready to add more complex functionality to include the integration of monitoring services or analysis tools. Newly the project can potentially be offered as a template for future additional services.

7.2 Future work

The practical project also has a great potential for future expansion. As we presented in the previous chapter, the improvement can be implemented in software communication and data transfer, where we can save up to more than ten times from the overall data exchange. The analysis also shows that communication between services produces many requests, which occasionally implies overheating. If we are able to reduce the number of requests, then we can possibly achieve a better performance.

The application is also prepared to separate whole frontend logic from the server-side and computational logic, which can help the project with clarity. Due to this separation, we can gain better awareness of which part should be resolved and where. It will avoid the technologies being mixed together.

Bibliography

[1] *Building Microservices with ASP.NET Core: Develop, Test, and Deploy Cross-Platform Services in the Cloud.* 1005 Gravenstein Highway North, Sebastopol, CA 95472 : O'Reilly Media, Inc., 1st edition, 2017. ISBN 1491961732.

[2] ANDYPI. *Cloud Computing Part 5: SaaS (Software as a Service)* [online]. [accessed 2020-06-01]. Available: <`https://andypi.co.uk/2016/05/23/cloud-computing-part-5-saas-software-as-a-service/`>.

[3] BEAZLEY, D. – JONES, B. K. *Python Cookbook.* 1005 Gravenstein Highway North, Sebastopol, CA 95472 : O'Reilly Media, Inc., third edition, 2013. ISBN 978-1-449-34037-7.

[4] BELL, L. et al. *Agile Application Security: Enabling Security in a Continuous Delivery Pipeline.* 1005 Gravenstein Highway North, Sebastopol, CA 95472 : O'Reilly Media, 2017. ISBN 9781491938799.

[5] BRENDRYEN, H. – DROZD, F. – KRAFT, P. A digital smoking cessation program delivered through internet and cell phone without nicotine replacement (happy ending): randomized controlled trial. *Journal of medical internet research.* 2008, 10, 5, s. e51.

[6] DOCKER. *About storage drivers*, . Available: <`https://docs.docker.com/storage/storagedriver/`>.

[7] DOCKER. *What is a container?* [online]. [accessed 2020-04-17]. Available: <`https://www.docker.com/resources/what-container`>.

[8] FOUNDATION., P. S. *The Python Profilers* [online]. [accessed 2020-05-26]. Available: <`https://docs.python.org/3.8/library/profile.html`>.

[9] GARCIA, D. F. et al. Experimental evaluation of horizontal and vertical scalability of cluster-based application servers for transactional workloads. In *8th International Conference on Applied Informatics and Communications (AIC'08)*, Page(s): 29–34, 2008.

[10] HENNESSY, J. – PATTERSON, D. *Computer Architecture: A Quantitative Approach.* ISSN. 225 Wyman Street, Waltham, MA 02451, USA : Elsevier Science, 2011. ISBN 9780123838735.

[11] INONIT.NO. Serafin framework. <`https://github.com/inonit/serafin`>, 2020. Last accessed 2020-03-28.

[12] KULHÁNEK, A. et al. eHealth Intervention for Smoking Cessation for Czech Tobacco Smokers: Pilot Study of User Acceptance. 2018, Page(s): 81–85.

[13] LANARO, G. *Python High Performance - Second Edition.* 35 Livery Street, Birmingham B3 2PB, UK. : Packt Publishing, 2nd edition, 2017. ISBN 9781787282896.

[14] LEIFER, C. *GitHub - Huey 1.2.* Available: <`https://github.com/coleifer/huey/blob/master/CHANGELOG.md`>.

[15] LOTT, S. F. *Modern Python Cookbook.* 35 Livery Street, Birmingham B3 2PB, UK. : Packt Publishing, 2016. ISBN 9781786469250.

[16] MA, L. – YI, S. – LI, Q. Efficient Service Handoff across Edge Servers via Docker Container Migration. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, SEC '17, New York, NY, USA, 2017. Association for Computing Machinery. doi: 10.1145/3132211.3134460. ISBN 9781450350877.

[17] NEWMAN, S. *Building Microservices.* 1005 Gravenstein Highway North, Sebastopol, CA 95472 : O'Reilly Media, Inc., 1st edition, 2015. ISBN 1491950358.

[18] PETERSON, B. *PEP 373 – Python 2.7 Release Schedule.* Available: <`https://www.python.org/dev/peps/pep-0373/`>.

[19] PROJECT, D. *Specifying which fields to save* [online]. [accessed 2020-06-14]. Available: <`https://docs.djangoproject.com/en/3.0/ref/models/instances/`>.

[20] RAMALHO, L. *Fluent Python: Clear, Concise, and Effective Programming.* 1005 Gravenstein Highway North, Sebastopol, CA 95472 : O'Reilly Media, 2015. ISBN 9781491946251.

[21] RICHARDSON, C. *Microservices Patterns: with examples in Java.* 20 Baldwin Road, Shelter Island, NY 11964 : Manning Publications Co., 2019. ISBN 97816117294549.

[22] S. ANITHA REDDY, e. a. Data Distribution Methods in Cloud Computing. *International Journal of Engineering Research & Technology (IJERT).* 2014, 03, 1.

[23] SANFILIPPO, S. *GitHub - Redis 3.2.* Available: <`https://github.com/antirez/redis/tree/3.2`>.

[24] WANG, M.-J. *Linked list* [online]. [accessed 2020-04-25]. Available: <`https://patents.google.com/patent/US7028023`>.

[25] ZIADE, T. *Python Microservices Development.* 35 Livery Street, Birmingham B3 2PB, UK. : Packt Publishing, 2017. ISBN 978-1-78588-111-4.

Appendix A

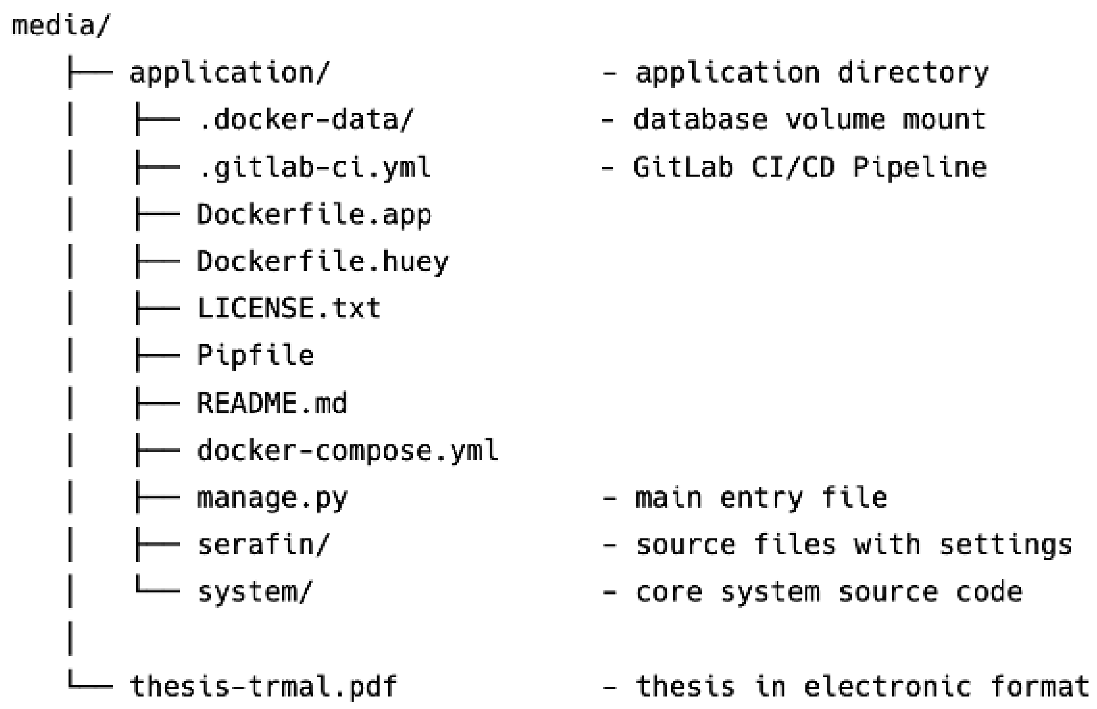
Nomenclature

API	Application Programming Interface
AWS	Amazon Web Services
BNF	Backus–Naur form
CD	Continuous Delivery
CI	Continuous Integration
CPU	Central Processing Unit
gzip	GNU zip
HTTP	Hypertext Transfer Protocol
IaaS	Infrastructure as a Service
JSON	JavaScript Object Notation
LTS	Long-therm Support
LXC	LinuX Containers
OS	Operating System
PaaS	Platform as a Service

RAM	Random Access Memory
SaaS	Software as a Service
SQL	Structured Query Language
VSC	Version Control System

Appendix B

Content of attached CD



```
media/
    ├── application/              - application directory
    │   ├── .docker-data/         - database volume mount
    │   ├── .gitlab-ci.yml        - GitLab CI/CD Pipeline
    │   ├── Dockerfile.app
    │   ├── Dockerfile.huey
    │   ├── LICENSE.txt
    │   ├── Pipfile
    │   ├── README.md
    │   ├── docker-compose.yml
    │   ├── manage.py             - main entry file
    │   ├── serafin/              - source files with settings
    │   └── system/               - core system source code
    │
    └── thesis-trmal.pdf          - thesis in electronic format
```