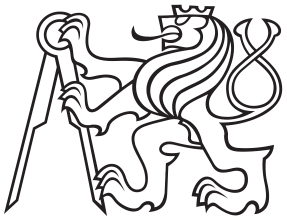
Master Thesis



Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Computers

Continuous Integration of web therapeutical application

Vít Říha

Supervisor: doc. Ing. Daniel Novák, Ph.D.
May 2024

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Říha Vít** Personal ID number: **465826**

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:

Continuous Integration of web therapeutical application

Master’s thesis title in Czech:

Continuous Integration webové terapeutické aplikace

Guidelines:

The topic of the work is a existing web application that enables
management of therapeutical programme. The application runs on Django python web framework with PostgreSQL as database and Huey/Redis to providing asynchronous multithread processing and scheduled tasks.
1. Study and analyze existing therapeutical application.
2. Optimize scheduling algorithm in Rust enviroinment
3. Implement and configure the resulting optimization
4. Test the best framework on the real environment containing at least
5000 users

Bibliography / sources:

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

Name and workplace of master's thesis supervisor:

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

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

Date of master's thesis assignment: **13.02.2023** Deadline for master's thesis submission: **24.05.2024**

Assignment valid until: **22.09.2024**

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

In the first place, I want to thank my supervisor, doc. Ing. Daniel Novák, Ph.D., for supporting me throughout the entire process of working on this thesis with an overwhelmingly positive attitude. Next, I want to thank Ing. Jindřich Prokop for helping me to get oriented in the project and providing basic technical support related to the original project. I also want to thank my closest family for their support and understanding when working on the thesis. I would like to dedicate this work to my beloved grandmother, who passed away on 11 March 2023.

V první řadě chci poděkovat mému vedoucímu práce, doc. Ing. Danielu Novákovi, Ph.D., za jeho podporu v průběhu celého procesu tvorby této diplomové práce s nesmírně pozitivním přístupem. Dále chci poděkovat Ing. Jindřichu Prokopovi za jeho pomoc zorientovat se v projektu a poskytování základní technické podpory vztahující se k původnímu projektu. Také bych chtěl poděkovat mé nejbližší rodině za jejich podporu a pochopení při tvorbě této práce. Tuto práci bych chtěl věnovat mé milované babičce, která nás opustila 11. března 2023.

Declaration

I declare that this work is all my own work and I have cited all sources I have used in the bibliography.

Prague, May 24, 2024

Prohlašuji, že jsem předloženou práci vypracoval samostatně, a že jsem uvedl veškerou použitou literaturu.

V Praze, 24. května 2024

Abstract

This work deals with optimisation of a therapeutical web application originally implemented in Python using the Django framework. Performance gains are achieved by reimplementing a key module of the application into Rust. Motivation for this are general performance limitations of Python as a consequence of its fundamental features. Rust, as a compiled language with emphasis on memory safety offers a promising alternative for performance-critical tasks. The work includes an analysis of the relevant parts of the original application, design and implementation of the Rust module, and its integration to the existing CI/CD solution. The results demonstrate a nearly 30% performance gain and significant improvement of its stability under heavy load. The work also provides an insight into the process of integration of Rust in web applications based on the Django framework and shows the potential for improvement of their performance, stability, and scalability.

Keywords: Python, Rust, PyO3, Optimisation, Django, Docker, CI/CD, DevOps

Supervisor: doc. Ing. Daniel Novák, Ph.D.
Na Zderaze 269/4,
120 00 Praha 2 - Nové Město

Abstrakt

Tato práce se zabývá optimalizací terapeutické webové aplikace, původně implementované v jazyce Python za pomoci frameworku Django. Zvýšení výkonu aplikace je docíleno přepsáním klíčové části aplikace do jazyka Rust. Motivací k tomuto kroku jsou obené výkonnostní limitace jazyka Python plynoucí z jeho základních vlastností. Rust, jakožto kompilovaný jazyk s důrazem na paměťovou bezpečnost nabízí slibnou alternativou pro úkony náročné na výkon. Práce zahrnuje analýzu relevantních částí aplikace, návrh a implementaci Rustového modulu a jeho integraci do existujícího CI/CD řešení. Výsledky demonstrují přibližně 30% nárůst výkonu aplikace a výrazné zlepšení její stability v zátěži. Práce také poskytuje náhled do procesu integrace Rustu ve webových aplikacích založených na frameworku Django a ukazuje potenciál zlepšení jejich výkonu, stability a škálovatelnosti.

Klíčová slova: Python, Rust, PyO3, Optimalizace, Django, Docker, CI/CD, DevOps

Překlad názvu: Continuous Integration webové terapeutické aplikace

Contents

1 Introduction 1

2 Analysis Of Existing Solution 3

2.1 Main Framework 3

2.2 Serafin 3

2.2.1 Application in the Project 4

2.3 Code Structure 5

2.3.1 Core Methods of the Engine Class 5

3 Designing Rust Solution 7

3.1 Comparison of Rust and Python . 8

3.1.1 Language Design 8

3.1.2 Performance 8

3.1.3 Static vs Dynamic Typing . . . 10

3.1.4 Developer Experience and Ecosystem 11

3.2 Choosing the Framework 12

3.2.1 PyO3 12

4 Implementing Rust Module 15

4.1 Implementation Process 15

4.1.1 Getting started 15

4.1.2 Common patterns between the original code and Rust implementation 19

4.1.3 Dynamic vs. Static Typing . . 20

4.1.4 Seemingly Compatible Data Types 23

4.1.5 Using Modules of the Original Implementation 24

4.1.6 Finishing up 25

4.2 Integration into CI/CD 26

5 Performance testing and optimisation 27

5.1 Tools and methods 27

5.1.1 Profiling 27

5.1.2 Performance/Load Testing . . 28

5.2 Chosen Strategy 30

5.2.1 Hardware 30

5.2.2 Designing the Test Plan 31

5.3 Running the Test 33

5.3.1 Preparation 33

5.3.2 Load Testing 34

5.4 Results 34

5.4.1 Incremental Load Testing . . . 35

5.4.2 Constant Load Testing 36

6 Conclusion 39

Bibliography 41

Figures

5.1 Codes Per Second, Incremental Load, Python implementation 35
5.2 Codes Per Second, Incremental Load, Rust implementation 36
5.3 Response Time Over Time, Incremental Load, Rust implementation 37
5.4 Response Time Percentiles, Constant Load, Python implementation 38
5.5 Response Time Percentiles, Constant Load, Rust implementation 38

Tables

5.1 Constant Load Testing Summary 37

Chapter 1

Introduction

Ensuring efficient and reliable performance is extremely important in today's web applications, as it could make the difference between the users staying, or leaving the website. This work deals with the optimisation of a web therapeutic application focused on smoking cessation, originally built with Python, by reimplementing a performance-critical module of the back end of the application in Rust.

Python, known for its simplicity and ease of use, often falls short in performance-critical scenarios due to its interpreted nature and dynamic typing. Conversely, Rust offers significant performance advantages with its compiled nature and strict memory safety guarantees. The work provides detailed comparison of Python and Rust programming languages, analysis of the original project with focus on the reworked module, options for integrating Rust into Python, as well as the caveats that were encountered during the implementation.

This work is not only relevant in the context of the smoking cessation web application. The same approach can be applied in a wide variety of web application implementations. Performance bottlenecks can severely impact user experience and engagement, making optimisation a crucial aspect of development. By reimplementing a key module in Rust, this project explores the potential for performance gains and improved resource management.

A core component of this work is the reimplementation process itself, which is documented to provide insights into the practical aspects of integration Rust with Python. This includes a discussion of the tools and techniques used, the challenges faced, and the solutions devised to overcome these obstacles. The integraion options for combining Rust and Python are examined, offering a direction for developers looking to adopt similar approaches in their projects.

The final chapters are devoted to analysis of the results of performance tests, and comparison of performance of the Python implementation and the

Rust implementation. These tests are designed to measure various aspects of the application's performance, such as response times, throughput and error rates. The findings from these tests are analysed to draw conclusions about the efficacy of using Rust for performance-critical components in web applications.

Chapter 2

Analysis Of Existing Solution

2.1 Main Framework

The application is build with Django as the main back end framework. Django is a high-level open source Python web framework developed between 2003 and 2005. Since then, it has grown into a robust framework used for building complex, database-driven websites.[1]

Django puts a lot of emphasis on security, which helps developers avoid many common mistakes by providing secure defaults and protection against vulnerabilities such as SQL injection, cross-site scripting, cross-site request forgery, and click-jacking.[1] This makes Django a reliable choice for rapid development of secure web applications.

2.2 Serafin

The project uses Serafin, which is a logic-driven content creation kit. It is a Django-based web platform that provides flexible building blocks for creating logic-driven websites, such as forms and questionnaires, self-help programs, e-learning programs, or dynamic websites with complex underlying logic.

Serafin is a therapeutic application developed by Inonit AS for SERAF, a Norwegian Centre for Addiction Research at the University of Oslo. It is designed to help users quit smoking while simultaneously gathering research data on the effectiveness of various therapeutic techniques.

Key features of Serafin include:

- **Built on top of Django:** Leveraging the power, flexibility and versatility of Django, Serafin benefits from a secure, well-supported, and highly scalable framework.

- **Graphical User Interface:** The platform includes a web-base GUI that streamlines the creation and management of content. This interface allows non-technical users to design and customise pages without needing to write code.
- **Logic-Driven Architecture:** Serafin's core strength is it's ability to handle complex logic. This is particularly useful for applications requiring conditional workflows, dynamic content generation, and personalised user experiences.
- **Invisible Special Actions:** Among the logic that Serafin provides for building the web application are also special nodes that provide functionality that is normally not visible to the users, such as sending cellular text messages and emails, or delayed actions.

The logic is organised in an oriented graph, or more specifically, a tree. A node typically represents a single page, but can also represent background actions. Unless it is a final page or action of the sequence, the node has one or more outbound edges. These edges may contain an expression that, when evaluated as `true`, represents the consecutive node.

There are two types of edges - special edges and normal edges. Special edges are always evaluated before the normal edges. It should be noted, that it is expected that only a single special, or a single normal edge evaluates to `true`, because the order of evaluation is not guaranteed. Once an edge evaluates to `true`, the consecutive node has been chosen.

2.2.1 Application in the Project

In this project, Serafin is employed to develop therapeutic sessions aimed at smoking cessation. The platform's web-based GUI is utilised to build a series of interconnected pages that guide users through these sessions. Each page within the series can contain various types of content and interactive elements, such as:

- **Forms and Questionnaires:** To collect user information and feedback, mainly for dynamic adjustment of subsequent content based on the user's responses.
- **Educational Material:** Providing users with information, tips, strategies and support for quitting smoking.
- **Interactive Exercises:** Engaging users in activities designed to reinforce behavioural changes and coping strategies.

2.3 Code Structure

The objective of this thesis is analysis and subsequent reimplementation of a performance-critical module of the application in a more performance-oriented programming language with the goal of achieving performance gains. A successful conventional optimisation was already performed by Jakub Trnal in 2020, resulting in a substantial reduction in computation time and demonstrating the effectiveness of the chosen strategies.[2] Building on this success, the current work aims to further enhance performance by leveraging the advantages of a compiled programming language over an interpreted language like Python.

Before starting with the implementation itself, it is beneficial to try to understand the original module. Upon first investigation, I realised that the Engine module, which is the subject of the optimisation, is always used in a single way. First, the constructor is used to get and instance of the class, and then a single method `run` is called. From this I deduced, that all other logic is effectively private to the class and that all of it is used exclusively internally in this class.

The `Engine` class interacts with several other modules of the project. Namely, it is several of the data access models, primarily for getting information about pages, session data, or getting and persisting user data. Another important dependency is the expressions module with its Parser class. This class is used to evaluate expressions that are often attached to edges. If the expression is evaluated to `true`, then that edge is used to traverse from the source node to the destination node.

2.3.1 Core Methods of the Engine Class

Contructor

The constructor accepts several arguments, but the most important one is `context`. It contains information about the data gathered in the previous step of the session, such as user's response to interactive parts of the session. This parameter is often crucial for determining what page will be displayed to the user next.

Run

The `run` method accepts two arguments: `next` and `pop`. If `next` is `true`, it means that the user is passing from one page to another during a session.

The id of the current node is extracted from the user's data and passed as an argument to `transition` method, from where the correct subsequent page is found and eventually returned. The `pop` parameter is used to finish up the current session when a final node of the session is reached. The current session is popped from the user's sessions stack and the subsequent session is initialised. If neither of the parameters is `true`, the current node gets "triggered". This performs the action according to the type of the node - for example an email node or an SMS node send an email or a text message, respectively. Other node types include: Expression node - sets a value on the user's record, or Page - sends data that should be displayed to the front-end.

Transition

`transition` is one of the core methods of the Engine class. It is the part of the algorithm that enables the transition from the current node to the next. It divides the edges directed from the current node into special and normal edges. Special edges are processed first. For each edge that the `expressions` module evaluates to true, the corresponding target node gets triggered. If any of the triggered nodes return a value, the method ends here and the same value is returned. Otherwise, a similar process is applied to the list of normal edges. If no traversable edge is found, typically at the end of a session, it is considered a "dead end" and the current session is popped from the user's session stack. Otherwise the result of `trigger_node` method is returned.

Trigger Node

`trigger_node` is a method that performs the action of the passed node based on its type. It typically calls a method from a different module that contains the logic to handle the action.

Chapter 3

Designing Rust Solution

First, I would like to summarise the ideas behind choosing to rewrite a part of the original project in Rust. Why should the Rust implementation be faster than Python?

1. **Compiled vs. Interpreted Language:** This is likely the biggest difference between the two languages. Rust is a compiled language, meaning its code is transformed into machine code before execution. This allows for various optimisations by the compiler, leading to faster runtime performance.

   Python is an interpreted language, which means its code is executed line-by-line by an interpreter at runtime. This introduces overhead, as the interpreter needs to process and execute each instruction on the fly.[3]

2. **Memory Management:** Rust employs a unique ownership model with strict compile-time checks for memory safety, which eliminates the need for a garbage collector. This results in more predictable and often faster memory management.[4][5]

   Python uses automatic garbage collection to manage memory, which can introduce latency during program execution as the garbage collector periodically runs to reclaim unused memory.[6]

3. **Type System:** Rust's static type system ensures type safety at compile time, which can prevent many runtime errors and allow for more aggressive optimisations at compile time.

   Python is dynamically typed, which introduces overhead for type checking and can lead to less optimised code paths.

3.1 Comparison of Rust and Python

Rust is a compiled systems programming language that aims to provide memory safety and concurrency without sacrificing performance. Developed by Mozilla and first released in 2010, Rust has gained significant popularity for its ability to produce fast, efficient, and secure code, making it an ideal choice for performance-critical applications. Rust's syntax is similar to C++ but with additional safety features, making it a modern alternative for systems-level programming.

Conversely, Python is a high-level, interpreted programming language known for its simplicity and readability. It emphasises ease of use, making it an excellent choice for beginners, prototyping, and rapid development. Python supports multiple programming paradigms, including procedural, object-oriented and functional programming. It has a dynamic type system and automatic memory management through a garbage collector. Python is widely used in web development, scientific computing, data analysis, artificial intelligence, and automation, thanks to its extensive standard library and the vast ecosystem of third-party packages. Python's flexibility and ease of learning have made it one of the most popular programming languages in the world.

3.1.1 Language Design

Rust and Python are designed with different philosophies and intended use cases, reflecting their core priorities and target audiences. Rust prioritises safety, concurrency, and performance. It employs a strict ownership model to ensure memory safety without a garbage collector, preventing issues like null pointer references and data racing. This makes Rust well suited for a variety of performance-critical applications.

Python, on the other hand, emphasises simplicity, readability, and ease of use. Its clean syntax and dynamic typing make it accessible to beginners and allows for rapid development across various domains, from web development to scientific computing.

3.1.2 Performance

Memory Management

Rust's ownership model is a key feature of the language, designed to ensure memory safety without a garbage collector. It enforces strict rules around

ownership, borrowing, and lifetimes, allowing developers to write concurrent programs without data races. Ownership Mensa that each piece of data has a single owner, which controls its lifetime. When the owner goes out of scope, the memory is automatically deallocated. This model eliminates common issues like null pointer dereferencing and double-free errors.[4][5]

In contrast, Python uses a garbage collector to manage memory. Python's garbage collector tracks object references and deallocates memory once an object's reference count drops to zero. It also uses cyclic garbage collection to detect and clean up reference cycles. This approach simplifies memory management for the programmer, but introduces overhead and can lead to unpredictable pauses in program execution.[6]

Rust's approach provides performance benefits by avoiding the runtime cost associated with garbage collection. Programs written in Rust can have more predictable performance, as memory deallocation occurs deterministically when variables go out of scope. However, this requires developers to be more mindful of how they manage references and lifetimes, which can increase the complexity of the code.

Python, on the other hand, trades off some performance for ease of use and developer productivity. Its automatic memory management allows developers to focus on the logic of their applications without worrying about manual memory management. This can be particularly beneficial in rapid development and prototyping scenarios.

Both models have their advantages: Rust's ownership model is well-suited for systems programming a performance-critical applications, while Python's garbage collector is ideal for applications where ease of use and development speed are more critical than raw performance.

Interpreted vs. Compiled

Compiled languages, such as Rust, are transformed into machine code by a compiler before execution. This process results in an executable file that the computer's hardware can run directly, leading to faster runtime performance, since the code is already translated into a low-level format. Additionally, compilation process ensures that errors are caught at compile time, providing robust type safety and reducing runtime errors.

Python, as a representative of the interpreted languages, executes code line-by line through an interpreter, which translates high-level code into machine code at runtime. This allows for greater flexibility and ease of debugging, but typically results in slower execution speeds compared to compiled languages.

This is also allows for dynamic typing, which further simplifies the source code, making it an ideal choice for scripting, web development, and rapid prototyping.

Another crucial difference, which has a big impact on performance, are compile-time optimisations. The compiler applies optimisations improving the efficiency and speed of the generated machine code. Rust, as a compiled language benefits significantly from these optimisations. Examples of the optimisations include: inlining functions, removing unused code (dead code elimination), and loop unrolling, all of which can lead to substantial performance improvements.

Python does not have a compilation step, and therefore cannot take advantage of these optimisations. However, implementations like PyPy support Just-In-Time (JIT) compilation, which can optimise code at runtime to some extent (typically after several executions of the section of the code).

3.1.3 Static vs Dynamic Typing

In statically typed languages like Rust, the type of a variable is known at compile time. The type of each variable must be manually declared and type checking is performed during compilation process. This allows for early error detection, because type mismatches are caught early during the compilation phase, reducing runtime errors and improving code reliability. It also enables possible optimisations made by the compiler, leading to overall better performance. It also enforces specification of parameter and return value types, which contributes to ease of use of external libraries and other code in general. However, static typing comes with disadvantages as well.It is generally more verbose, which expands the code and can make it less concise and less flexible, because the type of a variable cannot change at runtime without explicit conversions.[7][8]

In Python, the type of a variable is determined at runtime - this is called dynamic typing. Variables can hold any type of data at different times during execution, and type checking is done at runtime. Because variables can change types, programmers can produce more flexible and concise code. This is especially useful in scenarios where types are not known ahead of time. Less boilerplate is needed, as types do not have to be declared explicitly, making it easier to write and maintain code - particularly in case of scripting and prototyping.[7][8]

3.1.4 Developer Experience and Ecosystem

Learning Curve

Rust is known for its steep learning curve primarily due to its strict and complex type system, ownership model, and borrow checker. These features, while ensuring memory safety and concurrency, require developers to quickly grasp advanced concepts that differ from other, even low-level programming languages like C.[13][14][15]

Simplicity, readability, and ease of use are some of the biggest advantages of Python. Its syntax is clean and intuitive, which allows new developers to learn and start coding quickly. Python's design philosophy focuses on code readability and simplicity, lowering the learning curve significantly.[16][17]

Tooling and Development Environment

Rust's package manager and build system, Cargo, is very easy to use and integrate into development processes. It simplifies dependency management, compilation, and running tests, providing a seamless development experience. It reduces boilerplate code and setup overheads, making development more efficient.[13][15]

Python uses pip for package management and virualenv for creating isolated environments. While effective, these tools can be cumbersome, especially when dealing with dependency conflicts. Tools like pipenv and poetry have been developed to simplify these processes, combining package and environment management.[16][17][18]

Developers can take advantage of a wide range of Integrated Development Environments (IDE) for both of the languages. Visual Studio Code is one of the most popular options, especially for beginners, as it is free to use and offers both community-made and official extensions to enhance the programmer's experience. JetBrains offers two versions of their IDE for python - a Community Edition, which is free of charge, and supports Python's build tools, HTML, and even Rust via a plugin, which however, no longer receives support. The Professional edition adds support for a wide range of web development frameworks along with their build tools or database tools.[19] JetBrains also provides a standalone IDE focused entirely on Rust, called RustRover.[20]

3.2 Choosing the Framework

There are several frameworks available, that enable native execution of Rust code within a Python environment. Here is an overview of the primary options:

- **rust-cpython:** Framework which allows for writing a native Python module in Rust. It also supports using Python code within a Rust library. This dual capability makes it versatile for integrating Python and Rust functionalities.[21]
- **PyO3:** Originating as a fork of rust-cpython, PyO3 emerged during a period when rust-cpython was not actively maintained. PyO3 shares many features with rust-cpython but offers simplified memory management, making it easier to use. At the time of working on this project, PyO3 was also actively maintained, which ensured continuous updates and community support.[22]
- **CFFI and ctypes:** Python provides libraries that can load and call native C functions, and Rust is able to expose a C-compatible API using the Foreign Function Interface. This is basically a way of tricking Python into thinking that it is running a library written in C, while in reality it runs a Rust library.[24][25]

As a beginner in Rust programming language, I was looking for the option that would be the easiest to get started with. I chose PyO3, primarily due to its extensive documentation, and because it has the simplest memory management. This made the initial learning curve less steep and provided ample resources to troubleshoot and understand the integration process.

3.2.1 PyO3

PyO3 is a powerful framework designed to enable seamless integration of Rust with Python. It allows developers to write Rust code that can be called from Python, as well as to write Python extensions in Rust.

Key Features:

- **Ease of use:** PyO3 simplifies memory management, making it easier for developers to handle interactions between Rust and Python. It provides a simple way to convert between Python and Rust types, which reduces boilerplate code and potential errors in manual type conversions. For

example, Python lists can be converted to Rust vectors, or Python dictionaries into Rust hash maps, etc.[26]

- **Extensive Documentation:** PyO3 is well-documented, providing clear and detailed guides on how to set up and use the framework, The documentation includes examples and explanations of various features, helping developers quickly learn how to integrate Rust with Python efficiently.[22]

- **Performance:** By leveraging Rust's performance advantages, PyO3 allows developers to write high-performance Python extensions. This can be particularly beneficial for computationally intensive tasks where Python's performance might be a bottleneck.

GIL and Mutability. Python allows any object to be referenced and mutated from multiple references, facilitated by its use of boxed objects. To Global Interpreter Lock (GIL) ensures that only one thread can use the Python interpreter at a time, though non-Python operations can release the GIL. In PyO3, holding the GIL is represented by the `Python<'py>` token, which serves multiple purposes including providing global API access and ensuring that certain functions are called only while the GIL is held.

This token can be used to create Rust references that implicitly guarantee the GIL is held, which explains why some PyO3 APIs require the `py: Python` argument. For mutating operations on Python objects, PyO3 allows shared references, which can only be created with a GIL lifetime, instead of mutable Rust references.

Rust structs wrapped as Python objects (pyclass types) typically need mutable access. While the GIL ensures thread-safe access, PyO3 cannot statically guarantee the uniqueness of mutable references once ownership is transferred to the Python interpreter. This uniqueness is managed at runtime using `PyCell`, which is similar to Rust's `RefCell`. This runtime scheme ensures safe and controlled access to mutable references in a multi-threaded environment.[23]

Object Types. **PyAny** in PyO3 represents a Python object of unspecified type and is restricted to a GIL lifetime. PyAny can only occur as a reference (`&PyAny`) and is used when you need to access a Python object while holding the GIL. This is typically seen with intermediate values and arguments to pyfunctions or pymethods implemented in Rust, where any type is allowed.

The PyAny struct includes many general methods for interacting with

Python objects, such as `getattr`, `setattr` and `call`. These methods facilitate attribute access and method calling on Python objects within Rust.

Additionally, `&PyAny` types can be converted to their specific types using `downcast` and `extract` methods - the former *downcasts* the reference to a more specific PyO3 type, such as `PyList`, while the latter *extracts* a Rust type from the reference, such as `vector`.[23]

Using Python Modules in Rust. PyO3 allows calling Python functions from Rust using the `call` and `call_method` APIs. This integration makes it straightforward to execute Python functions, pass arguments, and handle return values directly within Rust.

For running existing modules from Rust, PyO3 offers the PyModule API. This is especially useful in cases, where existing logic needs to be called from Rust. Importing modules is straightforward using the `import` API. To call a method from an external module, we can combine these two approaches like this:

Listing 3.1: Calling functions from external modules

```
fn main() -> PyResult<()> {
    Python::with_gil(|py| {
        let builtins = PyModule::import(py, "builtins")?;
        let total: i32 = builtins
            .getattr("sum")?
            .call1((vec![1, 2, 3],))?
            .extract()?;
        assert_eq!(total, 6);
        Ok(())
    })
}
```

This will be crucial for the ability to call methods from the original Serafin implementation, as well as Django functionality.

Chapter 4

Implementing Rust Module

This chapter recounts my experience with rewriting a critical module of the application in Rust programming language. I will describe the implementation process - the difficulties and caveats. Next, I will outline the steps needed to integrate the module into the build stage of the CI/CD pipeline. Last, I will go over what I learned in the process and what could be further improved.

The reason for the decision to re-implement the module is fairly straightforward. The goal is performance optimisation, as Python is known for not being the most efficient programming language when in comes to performance. Oppositely, Rust is known for its exorbitant obsession with memory safety, as well as amazing performance.

4.1 Implementation Process

4.1.1 Getting started

I was considering adopting the test driven development approach, because the original module had some tests ready to run, but I soon realised, that I could not use these tests for this purpose, as they are not true unit tests. In the end I decided to go from the smallest prototype that I could test manually, for example just in Python console, and just try to replicate the code from the original module.

Learning PyO3

First, I needed to create the skeleton of a PyO3 project. This is a straightforward process. Using python package `maturin`, which can be installed using `pip`, one can initialise a PyO3 project in single step. The package generates both the necessary files. `Cargo.toml`, which is a Rust project definition file, that specifies project name, version, description, and dependencies, among

other things. The other generated file is located in newly created `src` folder, and it is called `lib.src`. The name of the file is important, because later when building the module, this is the filename that the compiler expects.

The bare `lib.src` file contains a simple test code intended as an introduction to the PyO3 framework, showing how to build a very simple function in rust and expose it to the Python environment. This is where I decided to experiment before diving into the documentation. At this point, I was a bit sceptical about the memory management. My goal was to verify that the generated Python module can accept an instance of a data structure and mutate its contents - as part of that rust code. First I prepared a function that accepts an instance of a list as a parameter and inserts an item to the list.

Listing 4.1: First encounter with PyO3 - file src/lib.rs

```
use pyo3::prelude::*;
use pyo3::types::PyList;

#[pyfunction]
fn add_to_list(list: &PyList) -> PyResult<()> {
    list.append("new item")?;
    Ok(())
}

#[pymodule]
fn PyO3_test(_py: Python, m: &PyModule) -> PyResult<()> {
    m.add_function(wrap_pyfunction!(add_to_list, m)?)?;
    Ok(())
}
```

The procedure to test the code was simple:

1. In Python, create an instance of an empty list.
2. Call the function on the Python module built from the PyO3 file.
3. An item is added to the list in the function of the module from the PyO3 file.
4. Use `print` to display the content of the list in the original Python script after the function call.

Listing 4.2: First encounter with PyO3 - file main.py

```
import PyO3_test

if __name__ == '__main__':
    list_instance = []
    PyO3_test.add_to_list(list_instance)
    print(list_instance)
```

I used `maturin develop` command to build the PyO3 module and inject it into the Python virtual environment of a test project and ran the python script. To my great surprise and pleasure, the output of this test was `['new item']` in the console. This was something that I hoped would not be an issue, but I had to verify this fact as soon as possible. This meant a couple of things. Most importantly, when an instance of an object gets passed as a parameter to my Rust module, the original Python context keeps ownership of the object. Also, the Rust function does not create a copy of the object when it receives it as a parameter. We can deduce this from the fact that when I called `print(list_instance)`, we can see the `'new item'` inside the list, which was added in the Rust module. This all might sound trivial, or unimportant, but it ultimately means, that I could rewrite the original module without changing the signatures of the original functions, which means that adopting the new module would just mean changing the import statement wherever the Engine module is used, because all the objects passed to it can in fact be mutated inside the Rust code.

Let's go over the structure of the Rust code in listing 4.1. First lines contain `use` statements, which is a way of importing different crates (Rust naming for modules). Then, we can see a macro `#[pyfunction]`. This is a way of signifying that this is a function that we will likely want to expose to the Python code the module will run in. The function `add_to_list` accepts a single parameter - a reference to a `PyList`. PyO3 comes ready with interfaces for all the basic data types that Python uses. `PyList` is the equivalent to `list`, and `PyDict` is the equivalent to `dict` [26]. My `add_to_list` function's return type is `PyResult<()>`. That is another type that is specific to PyO3. It is a shorthand for Rust's `Result<T, E>` type, where `E` implements `From<E> for PyErr`. This will raise a Python exception if the `Err` variant is returned. The `()` in my function signature's return type just means that the function does not return a value, but is fallible. This return type allows me to use the `?` on line 6. The return type of the method `PyList.append` also returns a `PyResult<()>`. The question mark operator unwraps valid values or returns erroneous values, propagating them

to the calling function. The next function uses the macro `#[pymodule]` which carries out exporting the initialisation function of our module to Python.

Starting work on Engine module

After I tried out some of the basic concepts of using the PyO3 framework, I began work on the Engine module by creating the constructor, which meant first defining class variables. A class in PyO3 is defined using two macros: `#[pyclass]`, which is created using Rust's `struct` statement, where the class variables are located. The other part of a class is the implementation of its methods. This is achieved using the macro `#[pymethods]` in combination with Rust's `impl` statement.

The original Engine module has many dependencies in the rest of the project. I needed a way to call methods on these different modules in the context of my Rust module. PyO3 offers interface to achieve this. This is a method that I wrote to be able to access the database using Django. It is used in the constructor, where the reference to the user can either be passed directly, or just a user id is passed, in which case the user needs to be retrieved from database.

Listing 4.3: Calling methods of other Python modules

```
fn get_user_by_id(py: Python, user_id: i32)
                    -> PyResult<PyObject> {
    PyModule::import(py, "django.db.transaction")?
        .call_method1("set_autocommit", (false, ))?;
    Ok(PyModule::import(py, "django.contrib.auth")?
        .call_method0("get_user_model")?
        .getattr("objects")?
        .call_method(
            "get",
            (),
            Some([("id", user_id)].into_py_dict(py))
        )?.into_py(py))
}
```

In Python, getting a user instance from database using Django is achieved in two lines of code, one of which is the import statement. The original code were two lines, each calling a function or method in an imported module, so technically four lines. That means that I first needed to import these two module. PyO3 offers an interface that can call any function and access any attribute on

any PyObject. This is because Python technically does the same, as it is an interpreted language, and type checks are performed in runtime. The painful part about this rewrite was the fact that every time I need to call a method on a non-standard object, I needed to insert an additional function call between the object and the function call or attribute. This can be seen in listing 4.3, where the original code called `transaction.set_autocommit(false)`, I needed to import the module and use `call_method1` function to call the `set_autocommit` method. The other call is even worse, as the original code accessed an attribute on the imported module, and only after that it called a method on that attribute. The code expands fairly fast as a result of these extra calls. The original Python module contains about 640 lines of code, many of which are docstrings, while the finished PyO3 module reached about 840 with no docstrings.

4.1.2 Common patterns between the original code and Rust implementation

On the rare occasion that the Python module checked for a `None` value, I got the opportunity to take advantage of Rust's `Option` object. Rust has a mechanism that ensures that whenever the developer cannot be 100% certain that a variable contains a non-None value, then it is visible on first sight. `Option` is an `enum` that has two values: `None` and `Some(T)`, where `T` is the data type of the value the `Option` can contain. This way, an optional argument `bar` of type `i32`, a 32-bit integer, of a function `foo` would be noted as `fn foo(bar: Option<i32>)`. This makes it clear, that `bar` might not contain a value. To access the contained variable, one can call `bar.unwrap()`. This is considered unsafe, as in the case where `bar` is actually `None`, Rust panics (the terminology for unexpected termination of the program). A different way to handle `Option`s is by calling `let bar = bar.unwrap_or(<default value>)`, which reinitialises the variable `bar` with type `T` and either the value contained in the `Option` in case it is `Some`, or the default value passed as a parameter in case the `Option` is `None`. Another two approaches to handling `Option`s are an `if let Some(value) = bar` along with an `else` statement, and a `match` statement.

Listing 4.4: Example handling of Option.

```
while !special_edges.is_empty() {
    if let Some(edge) = self.traverse(py, special_edges)? {
        // trigger the node...
    } else {
```

```
            break;
        }
    }
```

Listing 4.5: The original code in Python

```
while special_edges:
    edge = self.traverse(special_edges)
    if edge:
        # trigger the node...
    else:
        break
```

As an example of the `if let Some`, I present the code in listing 4.4, which is part of `transition` method in the Engine module. The return type of `self.traverse` is `PyResult<Option<PyObject>>`, which means that using the question mark operator, we effectively change the value to `Option<PyObject>` type. The `if` branch handles the case where `Option` is a `Some` value, and the `else` branch handles the `None` case.

4.1.3 Dynamic vs. Static Typing

Python's dynamic typing turned out to be a hard obstacle to overcome, as most of the original code did not have the (optional) typing annotations. To create a first prototype, I had to guess what type was used. Sometimes the type annotations were present, but they turned out to be more confusing than helpful. Let's take this snippet from the original implementation for an example:

Listing 4.6: Confusing type annotation in the original code

```
def init_session(self,
                 session_id: int = None,
                 node_id: int = None,
                 should_save: bool = True):
    session_id = session_id or self.user.data.get('session')
```

The first time that I saw this, I automatically assumed that `session_id` is an integer type, because it is a parameter of the function with a type annotation. So I was very surprised, when after running my Rust implementation, the code crashed inside this function. The error turned out to originate in the data type being stored inside the `self.user.data` collection. The `session` key of the `data` attribute contains a string and Python has no

problem assigning that value to the original parameter. I ended up fixing this issue by explicitly converting the value to Rust's `i32` type. I achieved this by creating a helper method, `any_to_i32`.

Listing 4.7: Any to i32 Method

```
fn any_to_i32(&self, data: &PyAny) -> Option<i32> {
  match data.get_type().name() {
    Ok("str") => {
      if let Ok(data_str) = data.extract::<String>() {
        let data_str = data_str.as_str();
        if "".eq(data_str) {
          return None;
        }
        return Some(data_str.parse().unwrap())
      }
      None
    },
    Ok("int") => Some(data.extract::<i32>().unwrap()),
    Ok(other) => panic!("Unexpected data type: {other}"),
    Err(_) => panic!("Failed getting type name...")
  }
}
```

In the end, I did not need a conversion from other than `string` data type. The method first gets the name of the data type of the variable that was passed as an argument. Based on the type name, it then converts the type to integer and returns it. In case of Python's `int` type, all that is needed is extracting the Rust's `i32` value. In case of a string, first a Rust `String` type is extracted from the `&PyAny`, and then the `i32` value is parsed from the string.

This was not the only time when I struggled with ambiguous data types. Consider the following snippet from the Python implementation:

Listing 4.8: Ambiguous data type of "edges" parameter in the original Python implementation

```
def traverse(self, edges: list, source_id: int):
    for edge in edges:
        expression = edge.get('expression')
        if expression:
            try:
                self.parser.refresh(self.user.data)
```

```
                passed = self.parser.parse(expression)
            except:
                passed = False
            if passed:
                return edge
        else:
            return edge
```

The datatype of the items of the `edges` parameter are not obvious at first glance. I did not notice this and the resulting code for this method was the following:

Listing 4.9: Ambiguous data type of "edges" parameter in the original Python implementation

```
fn traverse(&self, py: Python, edges: &PyList)
  -> PyResult<Option<PyObject>>
{
  let user_data = self.user.getattr(py, "data").unwrap();
  for edge in edges {
    if let Ok(expression) = edge.getattr("expression") {
      if let Ok(_) = self.parser
          .call_method1(
            py, "refresh", (user_data.as_ref(py), )
          ) {
        if let Ok(passed) = self.parser
            .call_method1(
              py, "refresh", (expression, )
            ) {
          if passed.extract::<&PyBool>(py)?.is_true() {
            return Ok(Some(edge.into_py(py)));
          }
        }
      }
    } else {
      return Ok(Some(edge.into_py(py)));
    }
  }
  Ok(None)
}
```

PyO3 allows iteration over `PyList` collections, however, the bindings will

have the general `PyAny` type. My assumption was, that simply by calling `getattr` on the `PyAny` object, I would get the value of `expression`. This, combined with the `if let Ok(...` expression and a safe `else` path led to a hidden bug, that manifested by not following the correct path in the decision tree based on user input. This was fixed by extracting the `&PyDict` type from the `edge` binding and calling `get_item` on that.

4.1.4 Seemingly Compatible Data Types

Another caveat of PyO3 are the seemingly compatible data type interfaces. A concrete example of this is Python's `OrderedDict` data type. Consider this short snippet of Rust code:

Listing 4.10: Example of PyDict data type usage in Rust

```
fn add_to_dict(dict: &PyDict) -> PyResult<()> {
   dict.set_item("key", "value")?;
   println!("Rust says: {:?}", dict.get_item("key"));
   Ok(())
}
```

It is a trivial piece of code that sets a value to a key in a reference to a python dictionary that was passed as a parameter. Now, let's call this code from a Python console.

Listing 4.11: Output when passing an instance of OrderedDict to the function from the previous listing in Python console

```
>>> from collections import OrderedDict
>>> import PyO3Test
>>> d = OrderedDict()
>>> PyO3Test.add_to_dict(d)
Rust says: Some('value')
>>> d
OrderedDict([])
>>> d.get('key')
'value'
```

We can see that the value is accessible through using the `get` method, but otherwise, neither the key, nor the value are visible to Python.

Listing 4.12: Output when passing an instance of the standard dictionary to the function from the previous listing in Python console

```
>>> d = dict()
```

```
>>> PyO3Test.add_to_dict(d)
Rust says: Some('value')
>>> d
{'key': 'value'}
```

This time, the key and value are visible immediately. This implementation detail caused a bug in the code, where adding values to an `OrderedDict` on the User instance did not work correctly, causing the newly added keys to not be visible later on, when the keys are accessed in a different module. To work around this issue, I used the `run` method of the PyO3 `Python` GIL interface, as seen in listing 4.13.

Listing 4.13: Workaround to set a key-value pair of an OrderedDict instance

```
fn ordered_dict_set(
        py: Python,
        dict: &PyAny,
        key: &PyAny,
        value: &PyAny
) -> PyResult<()>
{
    let locals = [
        ("d", dict),
        ("key", key),
        ("value", value)
    ].into_py_dict(py);
    py.run(r#"d[key] = value"#,None, Some(locals))?;
    Ok(())
}
```

This code essentially sets the value by running python code, using the standard notation `dictionary_instance[key] = value`.

4.1.5 Using Modules of the Original Implementation

On several occasions, it is necessary to call either part of the original implementation, that is located in another module, or part of the Django framework. To achieve this goal, I used PyO3's PyModule API. It provides methods for importing modules from the Python environment to be able to utilise them as if they were native Rust modules. This is done using he `PyModule::import` method. An example usage of this capability is demonstrated in the following listing:

Listing 4.14: Calling methods of other Python modules

```
let session_model = PyModule::import(py, "system.models")?
    .getattr("Session")?;
self.session = Some(session_model.getattr("objects")?
    .call_method(
        "get",
        (),
        Some([("id", session_id)].into_py_dict(py))
    )?.into_py(py));
```

First, the `system.models` module is imported. Then, the `Session` class is retrieved from the module using `getattr`. This would be equivalent to `from system.models import Session` statement in Python. This is saved in the `session_model` binding. After that, we need to call the `get` methods from the `objects` attribute, which is Django's data access layer. To achieve that, we use the `getattr` method and `call_method` methods, respectively. Notice the usage of `into_py_dict`, which converts Rust vector of tuples into a Python dictionary. This is then used as the keyword arguments for calling the `get` method. If we needed to pass any positional arguments, they would be represented by a Rust tuple as the second argument of the `call_method` method. Using the empty braces, we passed an empty tuple.

4.1.6 Finishing up

When most of the functionality was done, I started regularly running the tests that were prepared by the developers who worked on the project before me. This uncovered some bugs that were mostly fairly easy to fix. The process of building the module is following:

1. Use `maturin build --release` command to build the optimised binary `.whl` file.

2. Use `pip install` with the path to the `.whl` file to introduce the module to the Python environment.

After these steps, the Engine class can be imported into the project's modules using `from engine_rs import EngineRS`. Notice that I named the module slightly differently than the original one, so that in case we needed to revert to the original implementation, we could just rename the references used in the code to the original class, while it is apparent at first glance which Engine class is in fact being used.

4.2 Integration into CI/CD

To incorporate the build process into the CI/CD pipeline, I decided to add the procedure to the Dockerfile. The Dockerfile is a text document, containing the instructions required to build a docker image. This process is part of the build stage of the GitLab CI/CD pipeline. The build is composed of several steps:

1. Download `rustup`, a tool used to install rust compiler.
2. Run the `rustup` tool
3. Add `/root/.cargo/bin` to the `PATH` environment variable. This step is required, so that `maturin` can easily access the rust compiler.
4. Copy the Rust source files to the Docker container.
5. Run the command `maturin build --release`. This step builds the binary specific to the platform of the system on which the command is run. This is the reason why we do not build the binary locally - because this approach is more flexible, as it always builds the binary for the correct platform.
6. run `pip install <path to .whl file>`. This step is more complicated, because `maturin` does not support specifying the name of the output file. The name always contains information about the platform for which the binary was built, among other things. For example on my machine, the generated file's name is `engine_rs-0.1.0-cp38-none-win_amd64.whl`, where `win` in the name specifies Windows as the OS the binary is built for, and `amd64` the CPU architecture.

This finalises the process of installing the module on the Docker container. Unfortunately, despite the tests passing with no issues and the pipeline finishing with no problem, the code does not run properly on the target system. It is a great disappointment for me and the rest of the team, because I ran out of time to try and troubleshoot the issue before the deadline to submit this thesis. This also means, that regrettably, I cannot run any reasonable performance tests while the code is not implemented correctly.

Chapter 5

Performance testing and optimisation

This chapter focuses on the final stage of the project. We will explore the options I had for measuring the performance of the original and optimised version and present the results of the chosen method.

5.1 Tools and methods

5.1.1 Profiling

Profiling provides the most precise and detailed information about the performance of individual sections of a running program. It involves the analysis of a program to determine its runtime behaviour, specifically identifying which parts of the code are consuming the most resources.

- **CPU Profiling:** Assesses how the program utilises the CPU. Identifies sections of code that are computationally expensive.
- **Memory Profiling:** Analyses the memory allocation and usage during program execution. Helps detect memory leaks and optimise memory usage.
- **I/O Profiling:** Evaluates the input/output operations of a program, including dish access and network communication, to identify slow I/O operations.
- **Concurrency Profiling:** Analyses multi-threaded applications to detect race conditions or deadlocks.

Python profiling

As this project is dealing with a python implementation, I will elaborate on several profiling techniques specific to Python. These are important for

optimising code performance and identifying bottlenecks. I will cover various tools and techniques.

- **Deterministic Profiling** involves collecting precise data about the execution of a program by recording function calls, their execution time, and other statistics. The `cProfile` and `profile` modules in Python are commonly used for deterministic profiling.[9]
- **Statistical Profiling** collects data at specific intervals (sampling). Compared to deterministic profiling, it is less intrusive and introduces lower overhead, which is traded off for lower accuracy. `Yappi` is a typical tool used for statistical profiling.[9]

cProfile. `cProfile` and `profile` are built-in Python modules providing deterministic profiling. They record the number of function calls and their execution time. `profile` provides a bit more flexibility and configurability.[10]

pstats. `pstats` is a module used in conjunction with the two former tools to analyse profiling data. It allows for sorting of the results in various formats.[10]

line_profiler. `line_profiler` provides line-by-line profiling of Python code. It is useful for identifying which lines within a function are the most time-consuming.

memory_profiler. `memory_profiler` is used for memory profiling. It provides information about the memory usage on a line-by-line basis. It helps detect memory leaks and optimise memory usage.

py-spy. `py-spy` is a statistical profiler for python applications. It runs as a separate process, so it is non-intrusive and suitable for profiling running programs without modifying their code.

Yappi. `Yappi` is a statistical profiler that supports CPU-clock and time-based profiling. It also supports profiling of multi-threaded applications, making it a versatile tool for performance analysis.

5.1.2 Performance/Load Testing

This section explores the various tools and methodologies available for conducting performance and stress testing of web servers. It is a crucial aspect of web server maintenance and optimisation. It provides an insight into the server's ability to handle peak loads without compromising speed, efficiency,

or user experience. Using these tools and methods helps us evaluate a server's capacity to manage concurrent users, process requests quickly, and maintain stability under pressure.

Performance testing involves assessing the speed, responsiveness, and stability of a web application under a particular workload. It focuses on metrics, such as response time, throughput, and error rate. This type of testing helps in understanding the capacity limits of the application and in identifying the components that need optimisation.

Load testing, a subset of performance testing, specifically examines the system's behaviour under normal and peak load conditions. It simulates a large number of users accessing the application simultaneously to ensure that the back-end can handle high traffic without degradation in performance. Load testing is essential for applications expected to experience significant traffic spikes, such as e-commerce sites during sales events, or services providing lottery results, which get heavy traffic immediately after the announced result draw.

Key metrics in Performance and Load Testing include:

- **Response Time:** The time taken by the server to respond to a request. It is critical for user satisfaction, as longer response times can lead to user frustration, which in turn can lead to loss of the user's interest and diminished revenue.
- **Throughput:** The number of requests processed by the server per unit time. It indicates the capacity of the application to handle concurrent users.
- **Error Rate:** The number of failed requests compared to the total number of requests. High error rates under load can be indicative of stability issues that need to be addressed.

Tools for Performance Testing

Several tools are available to facilitate performance and load testing of web back-end, each offering unique features to address different aspects of testing.

JMeter. Apache JMeter is an open-source tool designed for load testing and performance measurement. It supports a wide range of protocols, including HTTP, HTTPS, SOAP, REST, GraphQL, and many more, making it versatile for testing various web applications. After running the tests, it provides a web report with processed results in the form of basic graphs and tables.[11]

LoadRunner. LoadRunner is a comprehensive performance testing tool that simulates thousands of users to test applications under load. It provides detailed analytics and reporting, helping identify performance issues and their root causes.

Locust. Locust is an open-source load testing tool that allows writing test scenarios in Python. It is highly scalable and can simulate millions of users to test the performance of web applications.

Gatling. Gatling is an open-source load testing tool primarily used for testing web applications. It is known for its high performance and ability to handle large-scale load tests efficiently.

5.2 Chosen Strategy

In this section, I will elaborate on the decision-making process for choosing the tool and strategy for performance testing of both the original implementation and the optimised version of the application. Choosing the right tool and strategy is critical in validation of the improvements.

My first consideration was profiling, as it provides accurate and easily comparable measurements of the improvements between the two versions. Since the optimisation targeted a single module, comparing the run times of the class's methods before and after optimisation would be most beneficial. However, this task proved challenging because the application runs in a Docker environment. This limited me to using less accurate statistical profiling, with the added difficulty of running it within the Docker instance.

Ultimately, it was much easier and more straightforward to use performance testing tools and flood the application with requests.

For this task, I chose Apache JMeter for its detailed documentation, versatile toolset, high performance, comprehensive reporting capabilities, and strong market adoption and community support. These attributes allowed me to design a test suite within a day, despite it being my first encounter with the tool. Furthermore, JMeter is an industry-standard tool for performance testing, providing valuable experience with a tool I am likely to encounter again in my career.

5.2.1 Hardware

I decided to take advantage of the existing environment on my laptop, running Ubuntu OS. It already had all the necessary configurations and tools set

up since it was the machine I used for development on this project. However, to access the application over the local network, I needed to make a minor configuration change. This involved adding the machine's local IP address to the Cross-Origin Resource Sharing (CORS) and Cross-Site Request Forgery (CSRF) allowed origins settings. By doing this, I enabled secure and authorised network access to the application, facilitating comprehensive performance testing from other devices within the local network.

Running the application back end on a laptop provided a unique advantage for performance testing. Laptops generally have more constrained resources compared to dedicated servers or cloud environments, such as limited CPU power, memory, and disk I/O capacity. This environment made it easier to push the hardware to its performance limits, which was particularly useful for demonstrating the effectiveness of optimisations.

Conversely, the machine simulating the clients was my desktop computer equipped with a powerful CPU. This meant that generating and sending multiple requests simultaneously was not an issue. The high-performance CPU could handle the computational load required to simulate numerous client requests without becoming a bottleneck. This reliability was crucial, because it guaranteed that the client machine's performance would not skew the results of the test.

5.2.2 Designing the Test Plan

First steps

As this was my first experience with the tool, I began with the simplest requests and gradually built upon them. I started by adding an HTTP Request element to the default Thread Group. I opened the app I was testing in my browser and navigated to the page I wanted to include in my Test Plan. Using the Network tab of the browser's developer tools as a reference, I constructed a basic request. When I ran the test, I received a 403 code (permission denied), which was not surprising.

The first obstacle I needed to address was maintaining a user's session. Initially, I used an HTTP Cookie Manager element, a JMeter tool for setting, storing, and attaching cookies to the requests sent out. I copied the session cookies from the developer tools' storage tab into the cookie manager. When I ran the test this time, I was pleasantly surprised at how easy it was to set up a basic working configuration, as I received a success code.

Next, I wanted to add a POST request. A single therapeutic session in the app takes place on a single web page using Angular on the front end. I

needed to simulate the user filling out a field and pressing the "Next" button. The application uses a POST request for that. Just like before, I used the developer tools to listen to the requests sent out by the app and replicate the POST request in a new HTTP Request element. This also yielded a successful response.

Automated login

I was aware that moving through the therapeutic session persists that information about the user in the database, which meant that I needed more user records - specifically one user for each thread that I would use to send out the requests. Otherwise the server would respond with error messages, because the threads would send requests containing data that is not expected in the current step of the session. This made me realise that the strategy of manually copying session cookies into JMeter was not viable for this scenario.

I started by creating the users in the app. Then, I made a comma-separated values (CSV) file containing the login information for each of the users. Using the CSV Data Set Config element, the data gets parsed in JMeter. Each thread gets a single line from the CSV, so each thread then represents a single user. I then used an Once Only Controller element to log into the users' accounts at the beginning of the test. However, one more element was needed to ensure a successful login. The login page contains a hidden input element containing a CSFR middle-ware token, which is a part of the application's security mechanisms. Logging in requires loading the login page, extracting the token from the body of the page and adding it to the POST request along with the user's credentials.

Finalising the configuration

Once I was able to log in with each thread seamlessly, I started adding the steps of the session. This was a fairly easy process of copying the request body from the browser and adding HTML Request elements for each of the requests. Finally, it was a question of configuring the number of threads and ramp up period according to the test I was conducting.

Resulting configuration:

- Thread Group
  - CSV Data Set Config (for loading user credentials)
  - HTTP Cookie Manager (for automatic session cookie retention)

- Only Once Controller (executes only once in the test)
  - HTTP Request (Get Token - loads the login page to extract the CSRF Middleware Token)
    - CSS Selector Extractor (uses CSS to extract the token from the hidden input element)
  - HTTP Request (Log In - POST request containing CSRF token and user credentials)
- HTTP Requests (11x - moving through the therapeutic session)
- Listeners for viewing the results when debugging the test

5.3 Running the Test

5.3.1 Preparation

To minimise interference from other software running on the laptop simulating the server, I ran the OS without Graphical User Interface (GUI). This should lower resource consumption from the operating system itself, which increases the performance, stability, and predictability. That is crucial for a consistent test environment.

As for the machine simulating the clients, I only resorted to turning off all unnecessary running software on the computer. As the machine is fairly overpowered for the task, I did not consider taking further steps. I validating my decision by performing an experimental run with the task manager open, monitoring system resources. The CPU load did not exceed 10%, and memory usage was nearly unaffected, so I was satisfied with the setup as it was.

However, I followed the instructions from the JMeter documentation and for the real tests, I ran the Test Plan from using the tool's CLI, with no GUI. I prepared a short batch script to create a new directory for each run, to streamline the process of collecting the data.

Listing 5.1: Batch script for running the Test Plan

```
@echo off
for %%i in (*.jmx) do set testrun=%%i
set folder=%testrun:~0,-4%_%date:~6,4%%date:~3,2%\
    %date:~0,2%T%time:~0,2%%time:~3,2%%time:~6,2%
md %folder%
md %folder%\web_report
jmeter -Jjmeter.reportgenerator.overall_granularity=10000\
```

```
    -n -t %testrun% -l %folder%\results.csv -e\
    -o %folder%\web_report
pause
```

The script finds a `.jmx` file containing the JMeter Test Plan definition, creates a directory with the same name, appending the current date and time to its name, runs the Test Plan, and saves the results in the newly created directory. It also sets a finer granularity for the graphs in the web report, because the default 60 seconds are too coarse for our data.

5.3.2 Load Testing

For the first test, the objective was to determine the maximum traffic the application could handle. To achieve this, I gradually increased the number of threads throughout the test, thereby increasing the server's load. I started with the original Python implementation and designed the test to add a new thread every 30 seconds, up to a total of 10 concurrent threads, and limited the test duration to 6 minutes. However, the server stopped responding within mere 2 minutes, eventually returning a timeout error. Despite multiple attempts, the results were the same. This was unexpected, so I performed another test with only two threads and a duration of 3 minutes. This also proved problematic, with the server regularly crashing within 30 to 120 seconds. It took about 10 attempts to get a run that did not crash.

Next, I repeated the procedure with the Rust implementation. It was a relief to find that this version was much more stable. The Rust implementation completed the first test with only minor issues, successfully handling most of the traffic, albeit with increasing latency and some errors. I ran the second test with the same configuration as for the Python implementation—2 threads for 3 minutes—to obtain comparable results.

The details of the test results are discussed in the following section.

5.4 Results

This section deals with the analysis of the collected data from the performance tests. It involves examining the results to identify key performance metrics such as response times, error rates, and system stability under different loads. By comparing these metrics between the original Python implementation and the Rust implementation, we can assess the effectiveness of the optimisations. This analysis provides a comprehensive overview of the performance of the application, highlighting the overall improvements achieved.

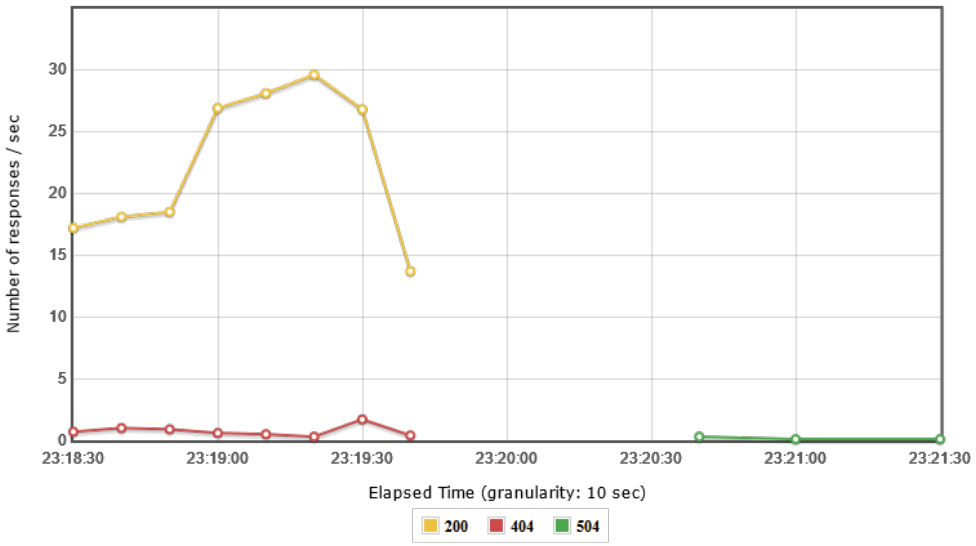


Figure 5.1: Codes Per Second, Incremental Load, Python implementation

5.4.1 Incremental Load Testing

The configuration for incremental load testing consists of gradually increasing the number of threads used to send requests to the server, starting with one thread and going up to 10 threads. The increment is made every 30 seconds to allow the server room for stabilisation. The ramp-up period is therefore 300 seconds, while the test is set to end after 360 seconds, or 6 minutes.

Python implementation

The original implementation did not do well during this test. After running the test several times, I finally gave up when the server endured for about one minute, after which timeout responses started appearing, as seen in figure 5.1. We can see throughput of about 18 successful responses per second (rps) in the first 30 seconds, when only a single thread was sending requests. After the initial 30 seconds, the second thread joins in and throughput peaks at 29.6 rps, before the server stopped responding. I cropped the graph from the right, as the later responses are all only 504 error codes and therefore are irrelevant.

Rust implementation

The Rust implementation endured the whole 6 minutes of the test, reaching the maximum load of 10 threads simultaneously sending requests to the server.

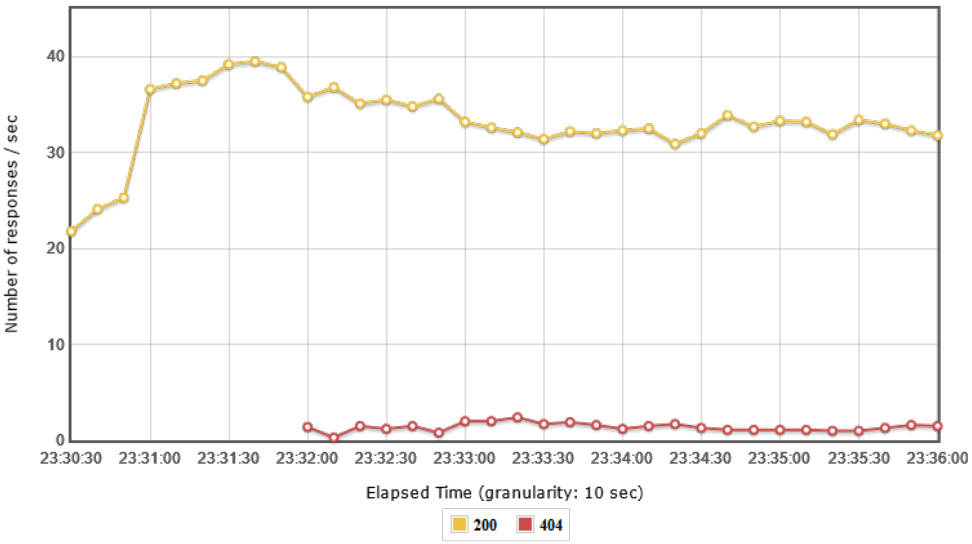


Figure 5.2: Codes Per Second, Incremental Load, Rust implementation

In figure 5.2 we can see a similar pattern to the previous figure, although the throughput in the first 30 seconds is higher, at over 24 responses per second (rps), with a single thread feeding the requests. The throughput peaks at 39.5 rps after the first minute of the test. When the fourth thread joins in, error responses start appearing, lowering the throughput of successful responses. However, the error codes took up at most 4.1% of the responses.

In figure 5.3 we can see how the increased load reflects on response times. Even under the heaviest load near the end of the test, the application handled nearly 90% of requests in less than 500 ms and 50% of requests in under 275 ms. Moreover, the response times appear to grow in a linear manner, indicating reasonable scalability and performance predictability under even higher load.[12]

5.4.2 Constant Load Testing

As the stability of the two implementations differs greatly, I decided to perform a test where the load on the server is constant and equal for both implementations, and compare the difference of latency of the responses between the Python and Rust implementations. This time, I configured the test to employ two threads right from the start, and let the test run for three minutes. Below is a summary of the results:

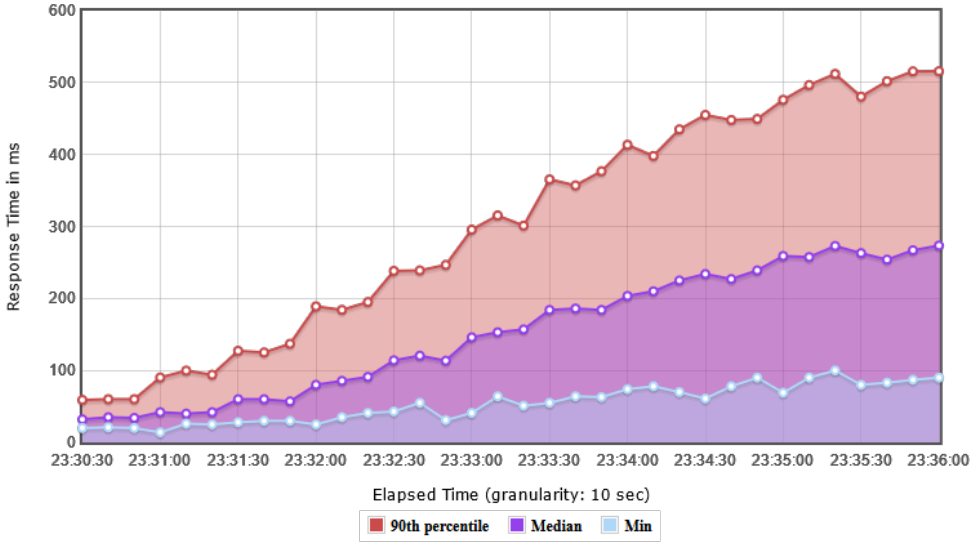


Figure 5.3: Response Time Over Time, Incremental Load, Rust implementation

Requests	Executions		Response Times [ms]			Throughput
Impl.	# Samples	Error %	Median	90th pct	95th pct	Trans/s
Python	4801	0.52%	52	143	178	26.67
Rust	6159	0%	44	99	128	34.23

Table 5.1: Constant Load Testing Summary

Python implementation

The original was unstable during this test, similarly to the first test. I took about 10 attempts to finish the three-minute test without the server becoming unresponsive. The response times seem to stay below 100 ms for most of the steps throughout the session, with the exception of the initial load and steps 1 and 10, as can be seen in figure 5.4. The throughput of this version of the server sits at 26.67 transactions per second.

Rust implementation

The superior stability of the Rust implementation can be seen in figure 5.5. Similarly to the original, steps 0, 1, and 10 are more problematic form the response time point of view. However, in the Rust version, up to 75% of those requests are processed in under 100 ms. It is not surprising that the throughput of this implementation is also superior, at 34.23 transactions per second, which means that it processes the requests about 28% faster.

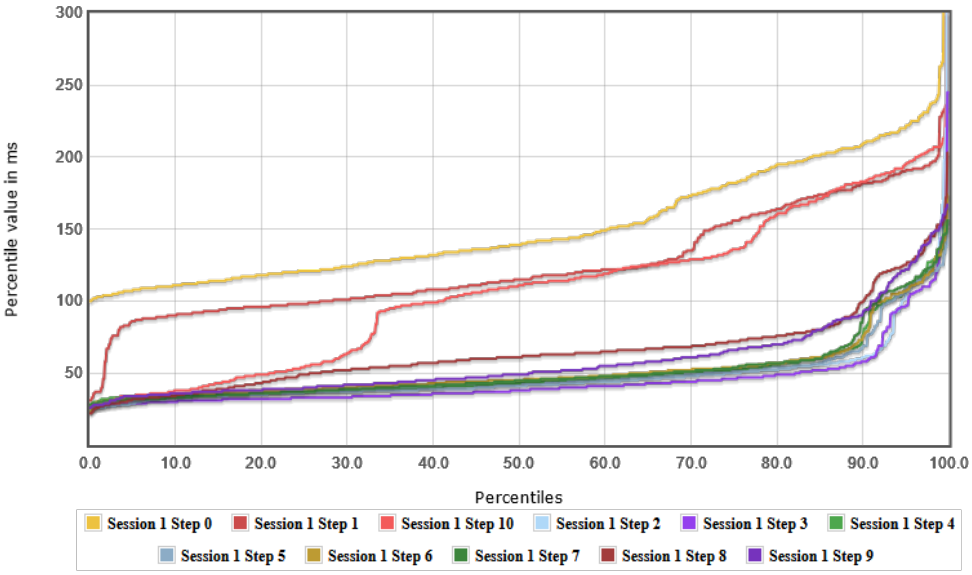


Figure 5.4: Response Time Percentiles, Constant Load, Python implementation

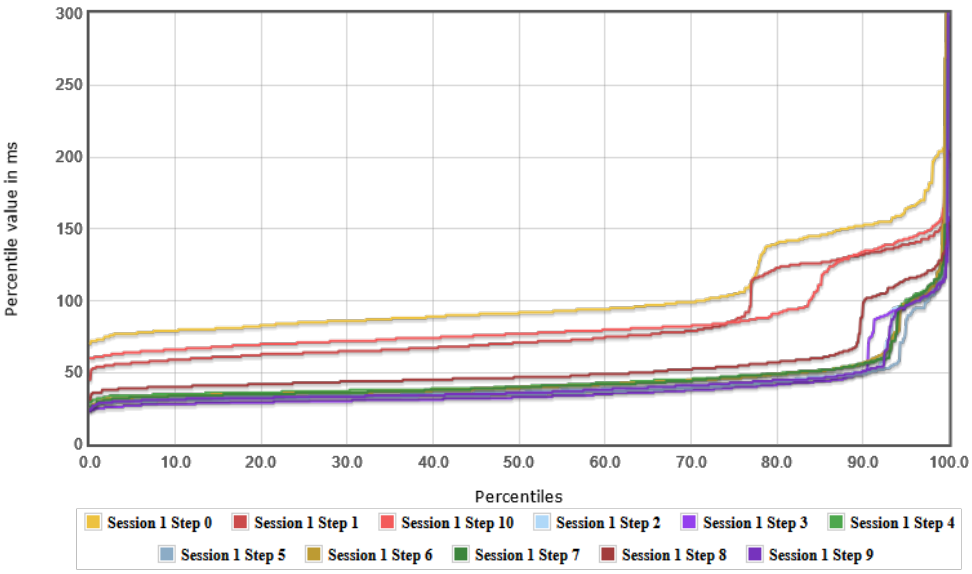


Figure 5.5: Response Time Percentiles, Constant Load, Rust implementation

Chapter 6

Conclusion

The optimisation of the therapeutic web application by transitioning a key module from Python to Rust has demonstrated nearly a 30% performance improvement. This was achieved by reworking only a relatively small part of the project, while the rest of it remained unchanged.

Along with performance gains, the project unveiled an unexpected positive side-effect in the form of greatly improved stability of the application under stress. The Rust implementation was able to process five times as much simulated traffic compared to the original application without the server becoming severely unstable. It should be noted, that higher load was not tested and real limits were not found.

This work underscores the potential benefits of leveraging Rust for performance-critical components in web applications, offering a direction for efforts of similar manner. The integration into the pre-existing CI/CD pipeline ensures that these enhancements are seamlessly incorporated into the development workflow.

Future work can explore further optimisation opportunities by expanding the use of Rust to other performance-sensitive areas - namely the `expressions` module, which is heavily used in the modified module and is a good candidate for potential reimplementation as well.

Bibliography

[1] MDN Web Docs. *Django introduction*. [ONLINE] Available at: https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Introduction. [Accessed 22 May 2024].

[2] Jakub Trnal. *Optimization of server solution and performance measurement*. Prague: CTU 2020. Master Thesis, CTU, Faculty of Electrical Engineering, Department of Computer Science.

[3] Free Code Camp. (2020) *Interpreted vs Compiled Programming Languages: What's the Difference?*. [ONLINE] Available at: https://www.freecodecamp.org/news/compiled-versus-interpreted-languages/. [Accessed 23 May 2024].

[4] Rust Documentation. *Understanding Ownership*. [ONLINE] Available at: https://doc.rust-lang.org/book/ch04-00-understanding-ownership.html. [Accessed 23 May 2024].

[5] Klabnik, S. and Nichols, C. (2019) 'Understanding Ownership', in *The Rust Programming Language (Covers Rust 2018)*. Updated for Rust 2018. San Francisco, CA 94103: No Starch Press, pp. 59–81.

[6] Python Developer's Guide. *Garbage collector design*. [ONLINE] Available at: https://devguide.python.org/internals/garbage-collector/index.html. [Accessed 23 May 2024]

[7] BairesDev Editorial Team *Static vs Dynamic Typing: A Detailed Comparison*. [ONLINE] Available at: https://www.bairesdev.com/blog/static-vs-dynamic-typing/. [Accessed 23 May 2024].

[8] Shannon Jackson-Barnes (2024) *Dynamic Typing Vs. Static Typing, Explained*. [ONLINE] Available at:

https://www.orientsoftware.com/blog/dynamic-typing-vs-static-typing/. [Accessed 23 May 2024].

[9] Antonio Molner. *Python Profiling*. [ONLINE] Available at: https://medium.com/@antoniomdk1/hpc-with-python-part-1-profiling-1dda4d172cdf. [Accessed 21 May 2024].

[10] Python Documentation. *The Python Profilers*. [ONLINE] Available at: https://docs.python.org/3/library/profile.html. [Accessed 21 May 2024].

[11] Apache Software Foundation. *Apache JMeter™*. [ONLINE] Available at: https://jmeter.apache.org/. [Accessed 21 May 2024].

[12] Alan Kay. *Chapter 1. Reliable, Scalable, and Maintainable Applications*. [ONLINE] Available at: https://www.oreilly.com/library/view/designing-data-intensive-applications/9781491903063/ch01.html. [Accessed 22 May 2024].

[13] BYBY. (2023) *Why Rust Has Steep Learning Curve*. [ONLINE] Available at: https://byby.dev/rust-learning-curve. [Accessed 23 May 2024].

[14] Team Gyata. (2024) *Rust Learning Curve*. [ONLINE] Available at: https://www.gyata.ai/rust/rust-learning-curve/. [Accessed 23 May 2024].

[15] Odgers, S. and Postolovski, T. *Why Rust is Worth the Struggle*. [ONLINE] Available at: https://www.cosive.com/blog/why-rust-is-worth-the-struggle. [Accessed 23 May 2024].

[16] MarketSplash Team (2024) *Julia Vs Python: A Comparative Analysis*. [ONLINE] Available at: https://marketsplash.com/julia-vs-python/. [Accessed 23 May 2024].

[17] Parvin Mohmad (2023) *Why Consider Python for Machine Learning and AI?*. [ONLINE] Available at: https://www.analyticsinsight.net/latest-news/why-consider-python-for-machine-learning-and-ai. [Accessed 23 May 2024].

[18] MarketSplash Team (2023) *Python vs Java: Main Differences & Best Use Cases (With Code Examples)*. [ONLINE] Available at: https://marketsplash.com/python-vs-java/. [Accessed 23 May 2024].

[19] JetBrains *PyCharm Professional vs PyCharm Community Edition*. [ONLINE] Available at: https://www.jetbrains.com/products/compare/?product=pycharm&product=pycharm-ce. [Accessed 23 May 2024].

[20] Kirill Smelov, JetBrains (2023) *Introducing RustRover – A Standalone Rust IDE by JetBrains*. [ONLINE] Available at: https://blog.jetbrains.com/rust/2023/09/13/introducing-rustrover-a-standalone-rust-ide-by-jetbrains/. [Accessed 23 May 2024].

[21] Daniel Grunwald. *rust-cpython on GitHub*. [ONLINE] Available at: https://github.com/dgrunwald/rust-cpython. [Accessed 23 May 2024].

[22] PyO3 User Guide. *Introduction*. [ONLINE] Available at: https://pyo3.rs/v0.18.3/. [Accessed 23 May 2024].

[23] PyO3 User Guide. *GIL lifetimes, mutability and Python object types*. [ONLINE] Available at: https://pyo3.rs/v0.18.3/conversions/types. [Accessed 25 May 2023].

[24] CFFI Documentation. *CFFI documentation*. [ONLINE] Available at: https://cffi.readthedocs.io/en/latest/. [Accessed 23 May 2024].

[25] ctypes Documentation. *ctypes — A foreign function library for Python*. [ONLINE] Available at: https://docs.python.org/3/library/ctypes.html. [Accessed 23 May 2024].

[26] PyO3 User Guide. *Mapping of Rust types to Python types*. [ONLINE] Available at: https://pyo3.rs/v0.18.3/conversions/tables. [Accessed 25 May 2023].