6CCS3PRJ
Optimising Server-Side Performance with IBM Instana & AI

Final Project Report

Author: Lavish Kamal Kumar

Supervisor: Dr Karine Even-Mendoza

Student ID: [redacted]

Programme Title: BSc Computer Science

[redacted]

Abstract

Performance bottlenecks can have material impacts on users of an application. The existing tools are limited and require careful analysis of method traces to debug performance issues. This dissertation introduces Aioptim. Aioptim is a command-line tool that identifies and resolves performance bottlenecks in Java and Python applications. This tool uses IBM's Instana APM, static analysis, and a fine-tuned binary classifier to identify method blocks contributing to performance degradation. These method blocks are replaced with AI-generated code through Ollama. The binary classifier had an F1 score of 0.813 on short/medium code samples. The classifier's performance declined on longer code samples. Qwen2.5-coder:32b-instruct was used for code generation and produced functional, performant code in more than 70% of the cases. This dissertation explores the benefits and limitations of using AI with dynamic/static analysis.

Originality Avowal

I verify that I am the sole author of this report, except where explicitly stated to the contrary. I grant the right to King's College London to make paper and electronic copies of the submitted work for purposes of marking, plagiarism detection and archival, and to upload a copy of the work to Turnitin or another trusted plagiarism detection service. I confirm this report does not exceed 25,000 words.

Lavish Kamal Kumar

Acknowledgements

I would like to extend my gratitude to my supervisor, Dr Karine Even-Mendoza for her unwavering support and guidance throughout the project. I would also like to thank my family and friends for their continued support and patience.

Contents

Chapter 1

Introduction

In modern times, performance has become a fiercely competitive industry, with companies fighting for seconds to ensure user traffic reaches them [14]. The consequences of considering performance as an afterthought are devastating to businesses. Amazon lost 1% in global sales per 100 milliseconds in delay [29]. It comes as no surprise that companies are allocating millions of dollars to save milliseconds [59]. All the while, software is becoming increasingly more complex, showing no signs of stopping soon [42]. The increasing complexity of software brings with it increasing costs [42]. Millions are being spent [59] on its prevention, yet the first responders to performance issues have technology that lags behind the users' needs. Increased software complexity, increased user demand and lagging technology are a potent mix for disaster.

The technology that is most commonly used today for performance tuning are profilers [55]. Profilers provide key information about specific functions and their metrics [55]. Profilers lack an end-to-end system that can detect bottlenecks and resolve them automatically. Such a system would eliminate the need to factor in human response times for time-sensitive tasks. To this end, this paper proposes a solution that combines traditional profiling techniques with artificial intelligence to build an automated end-to-end system to detect and resolve bottlenecks.

1.1 Objectives

The objectives of this project are to:

1. Leverage dynamic and static analysis to narrow the search space for inefficient endpoints.
2. Use deep learning and the identified inefficient endpoints to automate the identification of bottlenecks.
3. Harness generative artificial intelligence to automate the process of resolving the identified bottlenecks.

1.2 Report Structure

The report is broken down into the following structure:

1. The background covers the terms and concepts used throughout this dissertation.
2. The requirements and specifications outline the characteristics of the tool and describe how the functional and non-functional requirements will be satisfied.
3. The design chapter details the decisions involved in developing the tool.
4. The implementation chapter presents the concrete details involved in developing the tool.
5. The chapter on legal, social, ethical, and professional issues addresses the various concerns.
6. The evaluation chapter reflects on the advantages and limitations of the tool.
7. The conclusion summarises these concepts

Chapter 2

Background

This chapter aims to introduce background and context to the issue of performance bottleneck detection and resolution. The main topics covered in this chapter are the definitions, related works and a brief overview of the strategies.

Performance is a metric used to describe the efficiency of distributing compute resources. Historically, performance is described in conjunction with throughput and latency [18, 30]. Throughput defines how much processing happens concurrently, whereas latency describes the time it takes for the processing to occur [18, 30].

2.1 Performance Bottlenecks

In the context of application profiling, performance bottlenecks are identified as endpoints that have limited throughput and or higher latency. Endpoints are defined broadly as the intersection between the client and the machine.

2.1.1 Causes of Performance Bottlenecks

Performance bottlenecks are the result of:

i. Poor choice of data structures and/or algorithms [34].

ii. Resource exhaustion [34].

Traditionally, endpoints have been identified by evaluating low-level system metrics such as CPU utilisation [12].

As indicated previously, the scope of this research is to identify cases where there has been a poor choice of data structures and/or algorithms. Appendix A.1 contains a non-exhaustive list of popular performance issues.

2.1.2 Compiler Optimisations

Performance bottlenecks are not easily optimised by compilers; this is because the bottlenecks manifest themselves during the runtime of a program and, therefore, are not picked up by the static code analysis phase in the compilation process. Han et al. claim that performance issues manifest in 'specific inputs' or 'configurations,' which in turn trigger performance degradation [21].

There is an evident need for a strategy that captures the dynamic nature of bottleneck identification.

2.1.3 Chaos Engineering

One such strategy is chaos engineering. Chaos engineering is a set of processes that seek to simulate behaviour on an application that deviates from the norm [6]. An example of a chaos engineering technique is load generation [2].

Load generation is the process of exposing a system to a high number of requests [2]. In turn, this process manufactures the perfect conditions for exposing performance bottlenecks.

2.2 Dynamic Analysis

In isolation, load generation can not identify the endpoint causing the performance degradation. Profiling tools such as IBM's Instana are employed to supplement the load generation process [70].

Profiling is the process of gathering metrics related to the runtime performance of applications [70]. Tools like Instana use code instrumentation to provide fine-grained details. Often, these details include the endpoint latency. Due to programming language implementation differences, latency is not the most accurate metric for identifying bottlenecks [16].

2.3 Static Analysis

Static analysis delves deeper and explores these differences in implementation. Static analysis is the process of using rules to analyse code before execution [8].

An example of a static analysis technique is call graph construction. Call graph construction is the process of evaluating method calls to trace the journey of a request [53]. This approach can offer deeper insight into how slow requests are being processed by an application.

Finite rules, however, do not capture the infinite ways of producing inefficient code, nor do they consider the context [53]. This supports the claim that static code analysis is undecidable [32]. While practical solutions for static code analysis exist, assumptions are often made to meet the needs of commercial settings [25].

2.4 Hybrid Approach

It is evident that dynamic approaches lack low-level detail, and static approaches lack high-level context information. This hybrid approach employs the advantages of both techniques.

2.4.1 Related Work

Navas and Gehani's paper discusses the use of this hybrid approach to optimise memory usage in embedded devices [41]. Dynamic analysis collects information from user-provided parameters to inform static analysis phases [41]. This enables the OCCAM-v2 tool to remove redundant code [41].

Dynamatic is another tool that utilises this dual-phase approach [15]. Dynamatic's static analysis component instructs the dynamic analysis component to instrument code for the purposes of detecting race conditions [15].

Antal et al.'s study into bug prediction utilises static and dynamic analysis stages to generate a 'hybrid call-graph' [4]. Two metrics are extracted from these graphs to train a machine-learning mode [4]. This paper found that models trained with the 'hybrid call-graph' excelled in detecting bugs [4].

2.5 Machine Learning

It would be useful at this stage to consider an alternative. Machine learning is the process of learning patterns from distinct characteristics of data [54]. Trained models can make predictions on unseen data.

This study focuses on evaluating code snippets; however, there are numerous challenges to extracting relevant features from code. As is well known, it is not possible to calculate recursive depth accurately.

To this end, utilising traditional machine learning will not be satisfactory because it will exclude an important metric.

2.6 Deep Learning

An alternative to consider is deep learning. Deep learning is the process of learning patterns from unstructured data. The difference between machine learning and deep learning is that deep learning does not require feature extraction.

2.6.1 Transformer

An example of a deep learning architecture is the transformer. This is an architecture that enables sequence inference [66].

Bahdanau et al. introduced the concept of attention in their 2014 paper to solve the problem of recurrent neural networks not retaining long-term memory [66].

2.6.2 Transfer Learning and Fine-Tuning

Transfer Learning is an approach whereby a transformer model's existing knowledge is repurposed for a different task than it was initially intended for [28].

An example of a downstream task is classification. Classification is the task of assigning labels to data based on its features [61]. Binary classification is a subset of classification operating on only two labels. The labels assigned to a piece of data depend on the training set of a model.

2.6.3 Related Work

A technique exists to compare two pieces of code to probabilistically identify whether one code snippet will outperform the other [50]. Under the hood, this uses deep learning to compare the code's abstract syntax trees (AST) [50].

2.6.4 Evaluating Outputs

This section discusses the metrics used to evaluate the performance of a binary classifier. In the following, the term 'true positives' refers to the positive cases.

- Accuracy: proportion of correct predictions over all predictions [26]
- Precision: proportion of true positives over all predicted positives [26]
- Recall: proportion of true positives over all actual positives [26]
- F1 score: balance of precision and recall [26]

2.7 Generative Artificial Intelligence

Having explored the strategies used in bottleneck identification, this section analyses the techniques to resolve bottlenecks.

The term generative artificial intelligence describes models that have been trained on a large corpus of text and can draw 'statistically probable outputs' from their training [38].

2.7.1 Prompt Engineering

Prompt engineering is a field of study that researches how to constrain the output of generative models.

Information retrieval is a prompt engineering technique where additional domain-specific details are supplied to the prompt [46].

Context amplification is another technique where the models' attention is manipulated to focus more on a specific part of the prompt [46].

2.7.2 Related Work

Developer tooling is an example of a practical use of artificial intelligence. Systems such as Copilot and Cursor have rapidly gained notoriety for their ability to streamline the software development process.

2.7.3 Evaluating Outputs

Rosas et al. explore the idea of using generative AI to optimise code and seek to compare the generated code with the optimised code produced by a compiler [52].

This study concluded that generative artificial intelligence could optimise successfully with a high probability on small inputs [52]. However, models expressed 'non-deterministic' behaviour when exposed to longer inputs [52]. This highlights the need to evaluate the output code produced by a generative model.

2.7.4 Pass@K

One technique to evaluate the output code is 'pass@k' by Chen et al. [11]. This technique seeks to measure the probability of there existing one out of 'k' generated code solutions that can pass some predefined unit tests [11].

2.7.5 BLEU

'BLEU' is an algorithm that tokenises both the expected code solution and the code generated by an AI model. It then takes the token lists and compares their similarities [45]. This is quite a naive approach to identifying whether two pieces of code are similar because there could exist a function which does not use the same token sequence yet is semantically the same.

2.7.6 ICE-score

There exists a technique that uses large language models (LLM) to produce an 'ICE-score.' This score is based on the 'usefulness' and the 'functional correctness' of generated code [71].

2.7.7 CodeJudge

CodeJudge is a framework that guides an LLM in evaluating whether a piece of generated code meets a list of requirements.[64]

This adopts a dual phase approach to enable an LLM to evaluate its own understanding [64].

Chapter 3

Requirements & Specifications

This chapter outlines the system's functional and non-functional requirements and specifies how these requirements will be met.

3.1 Functional Requirements

The following outlines the characteristics of the system:

SFR1. Interface with GitHub to pull code from a repository.

SFR2. Interface with GitHub to push code to a repository.

SFR3. Collect endpoint information from IBM Instana.

SFR4. Parse methods in the provided source code.

SFR5. From the parsed code, identify code that contributes to performance bottlenecks.

SFR6. Generate performant source code.

SFR7. Validate the generated code.

From the perspective of the user, the system should enable the user to:

UFR1. Input details into the system.

UFR2. Receive optimised source code in the same repository.

UFR3. Authority over how performance bottlenecks are defined.

3.2 Non-Functional Requirements

The following is an outline of how the system should operate:

SNFR1. Detect slow code from parsed methods with at least 75% accuracy.

SNFR2. Enable developers to add language support in future works.

SNFR3. Automatically run at scheduled intervals.

From the perspective of the user, the system should aim to:

UNFR1. Keep users informed on the progress of optimisation.

UNFR2. Enable users to set up and operate the system with ease.

UNFR3. Be accessible.

3.3 Specifications

Table 3.1 outlines how the functional and non-functional requirements will be met while considering some assumptions.

Table 3.1: Assumptions and Specifications

Requirements	**Assumptions**	**Specifications**
SFR1 SFR2 UFR2	Fine-grained read/write access to repository	The tool can use the GitHub REST API
SFR3	Instana APM is properly configured & API keys provided	The tool can use the Instana REST API
SFR4	Source code files encoded in UTF-8	The tool can parse using the tree-sitter library
SFR5 SNFR1	User systems can run deep learning models	A binary classifier can be fine-tuned using a large dataset to make accurate predictions
SFR6	User provides access to cloud GPU	The tool can utilise the Ollama framework to generate code from pulled models
SFR7	User provides access to cloud GPU	A feedback loop mechanism can be created to validate AI-generated outputs
UFR1 UNFR2 UNFR3	The external systems are correctly set up	The tool can employ an accessible command line interface to enable users to input parameters
UFR3	The user is aware of their average endpoint latency	This can be another parameter for the command line interface
SNFR2	The source code for this tool adheres to best practices	The tool can be modularised and loosely coupled
SNFR3	The systems are continuously running	A loop structure can be implemented to support this
UNFR1	The prefect server is configured correctly	Users can have CLI and GUI access to monitor the tool

Chapter 4

Design

4.1 Introduction

This chapter begins with a high-level overview of the command-line tool's use case and architecture. Subsequent sections provide a detailed account of the components and the rationale behind design decisions, including evaluating alternative approaches. The design decisions build upon the functional and non-functional requirements discussed in the previous chapter. To conclude this chapter, the various deployment architectures are explored to highlight the tool's versatility.

4.2 Use Case

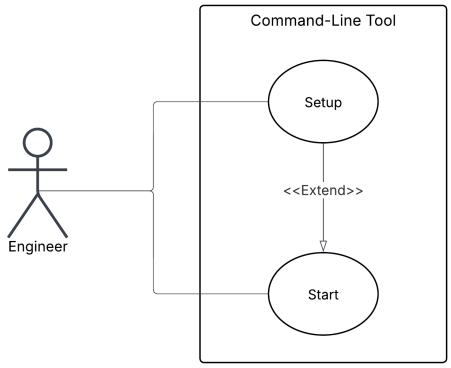

Figure 4.1: UML Use Case Diagram for the Command-Line Tool

Figure 4.1 illustrates a UML diagram.

- Setup involves configuring GitHub repository access, the Instana API key, and Ollama parameters.
- Start involves configuring maximum endpoint latency thresholds and run intervals.

A limited instruction set reduces the extraneous cognitive load [10] on the engineer administrating the tool. Cognitive load is a growing problem in the tech industry with the advent of new technologies at a breakneck pace. This problem is exacerbated by the fact that a large amount of software developers report being neurodivergent [63], which means they face a 'higher level of perceived extraneous cognitive load' [33]. Therefore, by implementing measures to reduce the load, such as limiting instruction sets and using colour [37] within terminal logging statements, the tool can be more accessible.

4.3 System Flow

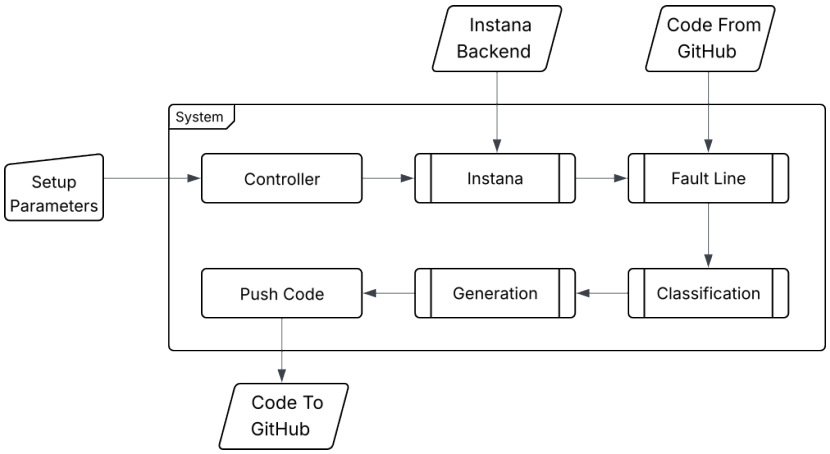

Figure 4.2: Flowchart for the System Flow, Based on the Pipes and Filters Pattern

The flowchart in Figure 4.2 has been employed to model the system architecture. A flowchart is a diagram that simplifies a process, using shapes to model the nodes in a system and arrows to illustrate their interactions.

The system operates in a pipeline-like fashion, known as the Pipes and Filters design pattern. This pattern consists of processing steps, referred to as 'filters', and the plumbing between

them, which are called ‘pipes’ [19]. A slight adaptation of this design pattern is used, whereby a singular controller node facilitates the communication between processes.

A significant advantage of this pattern is that it mimics the sequential nature of debugging performance bottlenecks in production code bases [13]. Another advantage is its natural adherence to the open-closed principle [40], which means further components can be easily appended to the pipeline and independently tested. A side effect of having independent components is loose coupling, promoting proper coding practices. A limitation of this approach is the sharing of context information between processing nodes [22].

A different approach could be to use the chain of responsibility design pattern. This design pattern relies on successive concrete handlers to transform the data or pass it forward to the next handler [69]. The principal limitation of this approach is that processing is not necessary at each stage of the chain; in other words, a handler can skip processing the data and pass it to the next handler [69]. However, the command-line tool relies on the output from previous stages to inform the inputs to the following stages. Therefore, a chain responsibility pattern does not align with the system’s needs.

4.4 Outline of Interactions

Assuming the setup and start parameters are valid, the controller instantiates the pipeline.

Slow endpoint information is retrieved from the Instana back end, and a fault line is generated by statically tracing the method calls made by the slow endpoint. The term ‘fault line’ refers to the call graph construction discussed in the background chapter.

By mapping the method traces, the system positions itself to capture the potential epicentre of the performance degradation. Owing to the fact that production source code is often highly cohesive, the fault line would consist of many code blocks. Therefore, a fine-tuned BERT model is employed to narrow the sample space by performing binary classification on the code blocks to separate ‘slow’ code from ‘fast’ code. The slow code samples are then passed to the generative model via Ollama. The generative model creates code that performs the same task much more efficiently. The optimised code undergoes a set of validation steps before it is pushed back onto the GitHub repository.

4.5 Instana Middleware

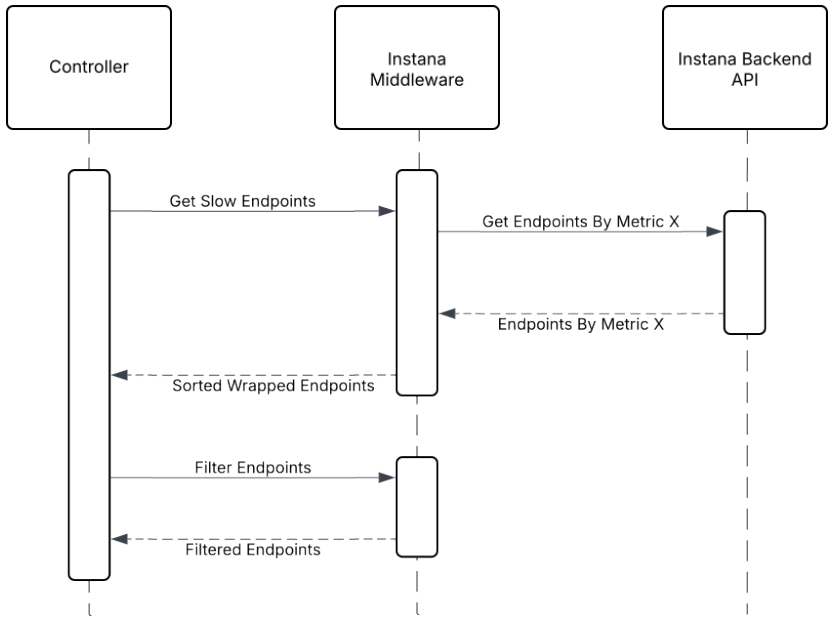

Figure 4.3: UML Sequence Diagram for the Instana Middleware

This section attempts to provide a detailed account of the Instana middleware using a UML sequence diagram (Figure 4.3). A UML sequence diagram is a tool that illustrates the exchange between entities [56]. Middleware is an entity that sits between two systems and facilitates their collaboration.

Per Figure 5, the Instana middleware sits between the controller and the IBM application programming interface (API). The API retrieves performance metrics from a preconfigured application perspective. Upon receiving the metrics data, the middleware wraps it into a usable object and returns it to the controller.

The controller then requests the middleware to filter the endpoints and pass them to the next stage of the pipeline.

As the sponsor of this dissertation, the scope of this study is limited to IBM's Instana application performance monitoring (APM) tool. One advantage of using Instana is that it comes with a preconfigured microservices application called robot-shop. This makes development and testing more efficient. Alternative APMs include Datadog, Elastic and Sentry. These APMs also provide endpoint performance metrics.

4.6 Fault Line

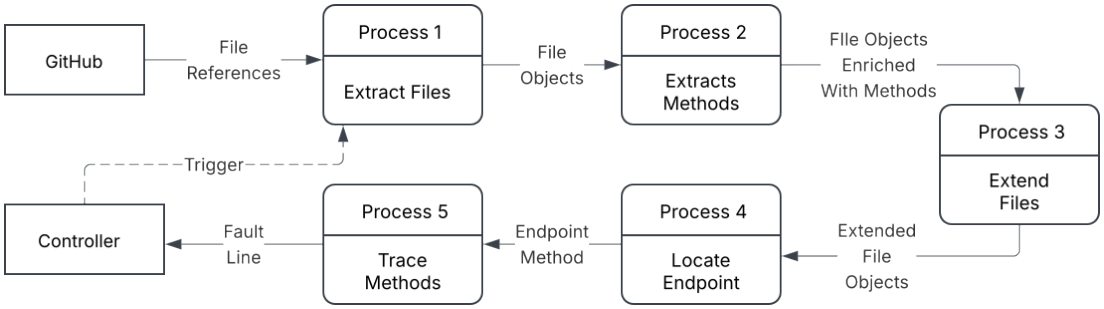

Figure 4.4: Data Flow Diagram for the Fault Line Generation System

The term 'fault line' is generally understood to have geographical connotations. Throughout this dissertation, however, a fault line will refer to the set of methods that can be traced from method calls made by slow endpoints.

Table 4.1: Method Calls Made by Each Method

Method Calls	Method 1	Method 2	Method 3	Method 4	Method 5
Method 1					
Method 2	✓				
Method 3		✓			
Method 4			✓		
Method 5	✓				

Table 4.1 illustrates the method calls in a sample program. To generate a trace, one must follow the method calls in a breadth-first manner.

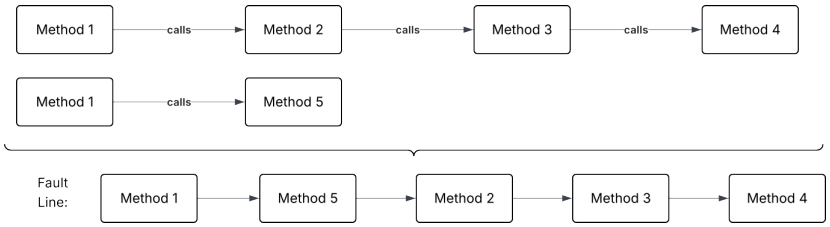

Figure 4.5: Fault Line Generated by Tracing Method 1

Production code bases are often made up of many highly cohesive components. As a result, numerous possible code blocks could contribute to poor endpoint performance. The purpose of fault line generation is to reduce the attack surface by exhaustively collecting methods involved with processing a request. This component achieves this by operating like a lightweight parser.

To combat the complex problem of static tracing, source code files must undergo several trans-

formative steps, as Figure 4.4 indicates. A data flow diagram is a tool that models the journey of data through a system [67].

Before applying transformative steps, the source code must be fetched from the deployment branch of the GitHub repository. Once the files are loaded, the methods in each file are extracted. Moreover, files may include import statements. Therefore, they are connected and enriched by the methods available in the imported files. This enables the exhaustive search across source code files.

The method associated with the slow endpoint (from the previous stage) is located in the graph-like file-method structure. Using the technique ascribed in Figure 4.5, a fault line is generated from method 1. This approach follows the exploratory discussion in the background chapter, where dynamic monitoring tools were employed in tandem with static analysis tools such as this component.

A design choice considered when developing this component involved selecting the medium from which to fetch files. GitHub was chosen because of its dominant position in the version control systems market [31]. However, a principal limitation of using GitHub is the inadequate API documentation. By contrast, competitors such as BitBucket and GitLab offer similar services with much more intuitive documentation.

Another design choice was adopting a lightweight parser to produce method traces. This was used since Instana can not provide low-level insights on performance issues. In practice, several techniques have been developed to trace methods. A prominent example in the field is Python's 'settrace' method [43]. This functionality can be used to keep track of methods involved in processing a request. Often, this is more accurate because it follows the exact execution path at runtime, whereas the current solution operates on a more exhaustive, naive approach. A caveat to using runtime tracing methods is the requirement to adapt the base source code. Therefore, the trade-off is made between the ease of use and the accuracy of the system's responses. A requirement of this tool is to solve the issue of performance bottleneck detection in an accessible manner. Furthermore, closer inspection of the runtime tracing approach indicates that it is not scalable, as new frameworks would need to adhere to a strict set of tracing constraints in order to be compatible with this tool. Weighing the trade-offs, this paper believes the current lightweight parser solution is the best approach forward.

4.7 Code Classification

This section is divided into two subsections. The first subsection describes the design for the training process, and the subsection that follows discusses how the classifier integrates into the pipeline.

4.7.1 Classifier Training

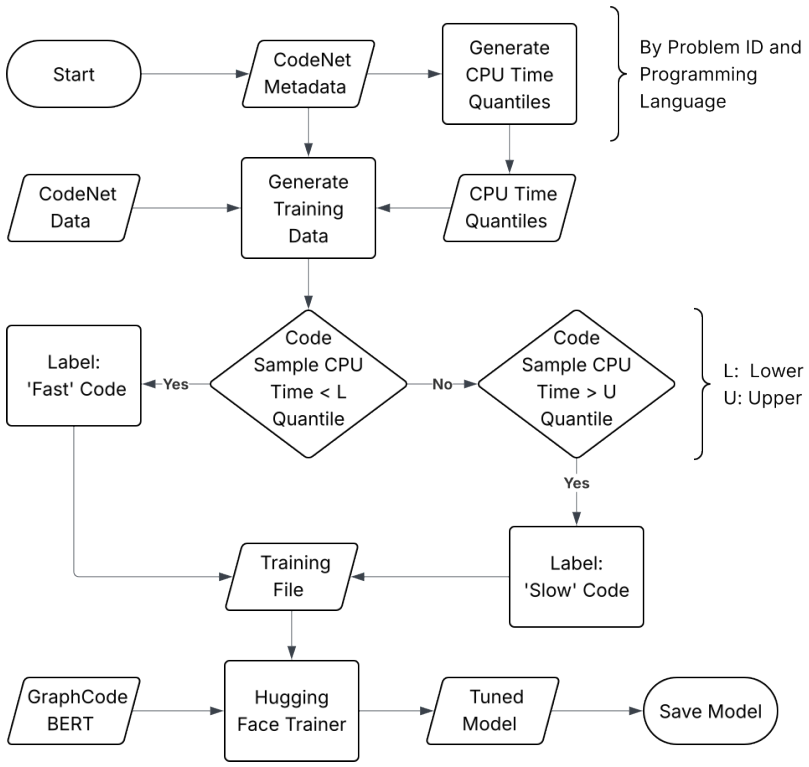

Figure 4.6: Flowchart Describing the BERT Fine-Tuning Process

Figure 4.6 describes the training process. The training set for this tool was IBM's CodeNet dataset [49]. This dataset consists of code submitted to online code-judging platforms. These platforms compare competitive programmers based on their code solutions' speed and memory usage. This dataset was utilised because of its extensive metadata and code samples. Another advantage of using this data is that it is naturally augmented. This means that many code samples differ slightly and, therefore, 'artificially increase the training set' [3]. This means the model generalises better to unseen data.

Microsoft's GraphCodeBERT model was selected for its awareness of data flow [20], which is

paramount to understanding the semantics of code. This model must be fine-tuned using the training set to perform binary classification on a code sample. Since GraphCodeBERT is aware of programming languages and their structure, this greatly simplifies the fine-tuning process.

A training set is required to fine-tune the model. The training set comprises data and its corresponding labels. Labelling the data beforehand would be a fruitful strategy to limit billed GPU usage during training. Labelling the data requires evaluating the metadata for each code sample. Code samples are grouped by the problem they are solving and the language in which the solution is implemented. Since this study employs CPU time as a metric to classify 'fast' and 'slow' code, a lookup table of CPU time quantiles is created. This table has entries for each problem and each language implementation.

Figure 4.7: Code Sample Labelling

Owing to the fact that the measurement of CPU time could be inaccurate [23], this introduces some uncertainty. To minimise the impact of inaccuracies with CPU time measurements, code samples that lay close to the central quantiles should be marked as ambiguous and disregarded. This is because those code samples could be categorised as fast and slow code based solely on sub-milliseconds differences in CPU run time. This partition creates two distinct zones for the labelling of fast and slow code based on CPU time quantiles, as illustrated in Figure 4.7.

Following the creation of a lookup table, a training set is generated based on the upper and lower quantiles. The final stage of training comprises fine-tuning the GraphCodeBERT model using the training set.

4.7.2 Classifier Integration

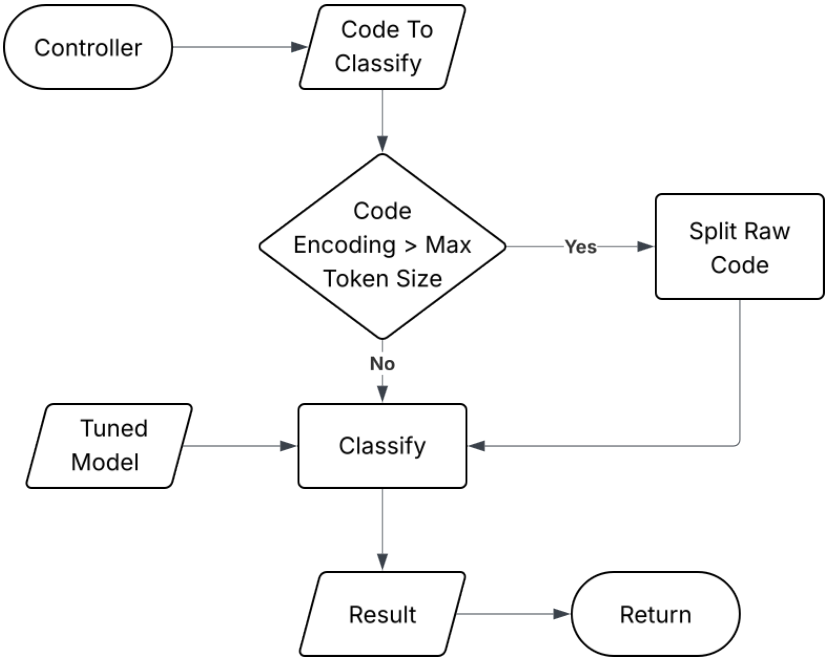

Figure 4.8: Flowchart Describing the Classification Process

The fine-tuned classifier is then integrated with the rest of the pipeline (Figure 4.8). Code extracts from previous stages are passed in and classified based on the model's training. An edge case is when code extracts exceed the maximum input length for the tokenisation process. With the GraphCodeBERT model, the tokeniser has a maximum input sequence length of 512 tokens. Input sequences that exceed this maximum are split into segments and processed individually. The mode of the resulting segment classifications is taken. The list of method classifications is returned to the controller.

4.8 Code Generation

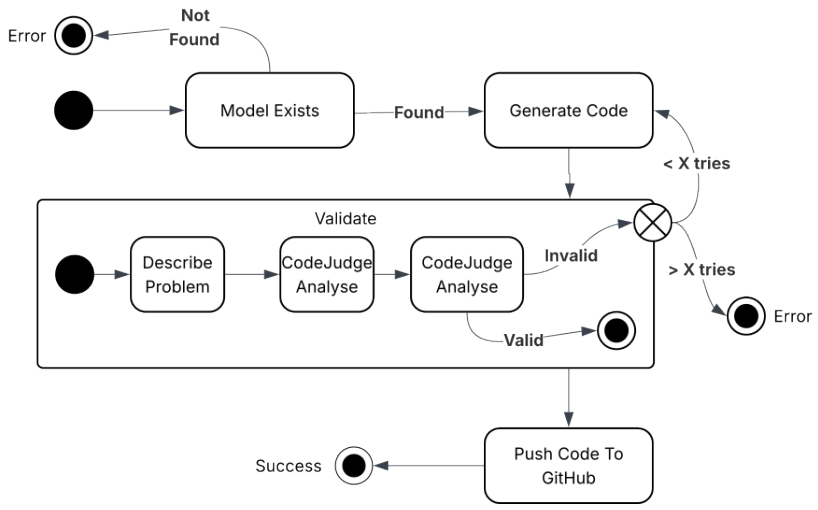

Figure 4.9: State Machine Diagram to Describe the Code Generation Process

Once the classification phase has identified the slow methods, the code generation phase aims to resolve those issues using generative artificial intelligence.

This project component utilises generative AI to develop valid and performant code. To this end, there are a number of transitional steps (Figure 4.9) to ensure that the generated code meets these requirements. A state machine is employed to represent these transitions. State machines represent the system's current mode and the possible modes a system can reach based on some conditions. The Ollama platform is leveraged to speed up the development and use of AI models. Ollama provides unified access to various AI models, enabling flexibility and streamlining development. API requests are sent from this tool to the user-provided server to poll the generative AI models. If the models the user has specified are available, the multi-stage code generation process begins.

At the beginning of the process, a request is made to generate more performant code, given an extract of slow code. The output from this phase is passed through a sequence of prompts, also known as prompt chaining. [27] In practice, sequencing prompts in such a manner has increased the generated responses' accuracy [27].

CodeJudge is used to validate the responses, as described in the background chapter. The design of the prompts maps perfectly to the CodeJudge 'Analyse-then-Summarise' framework [64]. This technique guides a model in making some evaluations and then summarising those

evaluations. More concretely, this tool adopts CodeJudge to ensure the generated code has the same functionality as the slow code being replaced.

If the generated code meets the requirements, it is pushed onto the GitHub repository in a new branch. If the generation framework cannot satisfy the requirements and exceeds the number of attempts, X, no code is produced and shipped.

The decision to use CodeJudge is clear, as other evaluation techniques described in the background chapter are naive (BLEU, BERT) or rely on the existence of complete unit tests (pass@k). This work makes another decision to limit the generation attempts to conserve GPU usage. This is a key consideration, as the models listed in this paper may require access to expensive computing infrastructure.

4.9 Deployment Architecture

To conclude this chapter on design, I will explore the deployment environments using deployment architecture diagrams. These diagrams illustrate the hardware and software configurations supported by this tool.

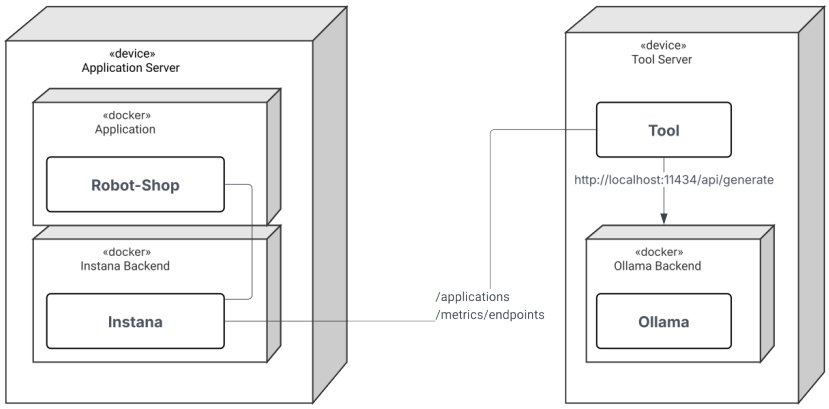

Figure 4.10: Deployment Diagram for Minimal Environment

Under the assumption that the user wants to configure the robot-shop application provided by IBM, there is a strict requirement that the Instana agent and the application backend reside on the same server. This is to enable endpoint discovery and collect system metrics. The configuration where the service and the tool reside on the same server (Figure 4.10) is cost-effective and increases the speed of the AI model inference. This is because the tool will not

need to communicate over a network. The trade-off for this design is introducing a single point of failure.

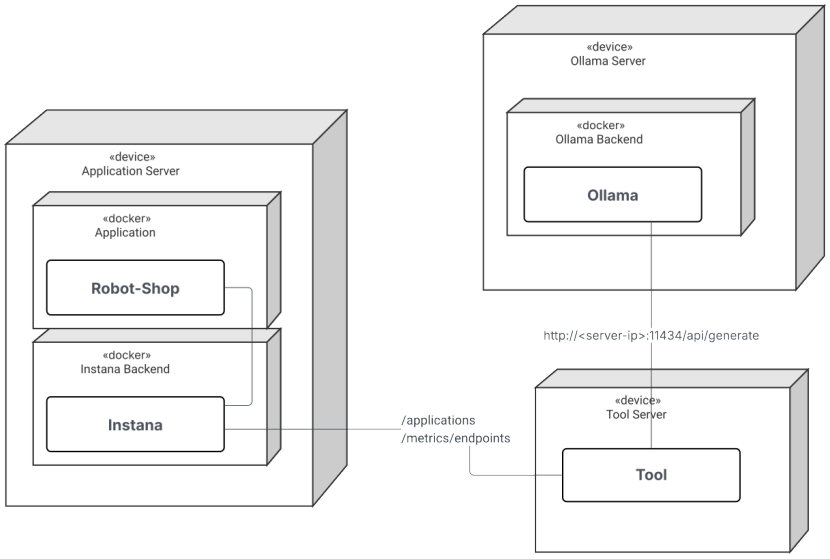

Figure 4.11: Deployment Diagram for Full Environment

An alternative approach is to host the Ollama service and the tool on different machines (Figure 4.11). This increases the resiliency of the infrastructure at a material cost. Also, because communication now must occur over a network, this introduces a source of failure.

Taking into account the various trade-offs, from a design perspective, it would be efficient to host the service and the tool on the same machine. However, this tool must be designed such that it can support both of these deployment models.

A limitation of utilising Instana is that running the tool, service, backend, and agent on the same machine, while possible, is not recommended. This is because the Ollama service could interfere with the proper functioning of the Instana agent.

4.10 Summary

To summarise, this chapter has explored a high-level overview of the tool's design and the components that comprise it.

There has also been a discussion on the design decisions made for each component and how they satisfy the requirements outlined in the previous chapter.

Chapter 5

Implementation

5.1 Introduction

This chapter aims to extend the work of the previous design chapter by providing concrete implementation details. This chapter is subdivided into five sections:

- Development approach
- Hardware, software and external libraries
- Component-wise implementation
- Further enhancements, challenges and limitations
- Testing

5.2 Development Approach

Developing tools of this calibre greatly benefits from a compatible development strategy. This implementation adopts a feature-driven development (FDD) approach. This approach builds on the agile methodology and seeks to dissect projects based on the features that need implementing [48]. Addressing features in this systematic way incentivises modularity and reduces feature creep. To this end, this system is implemented in a piecewise approach that complements the pipeline described in the previous chapter.

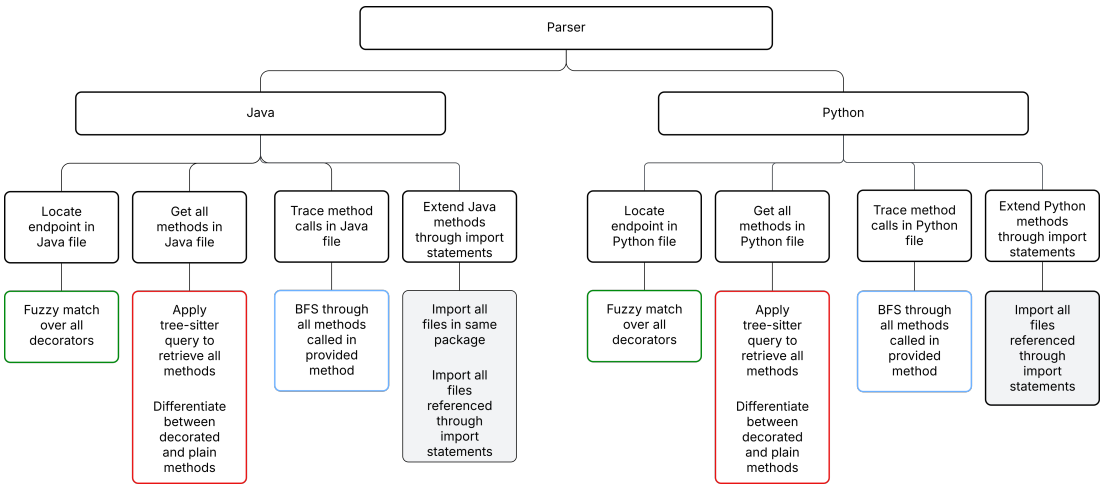

Figure 5.1: Break Down of Parser Components

Owing to the complexities involved in engineering solutions for each feature, there is a critical need to break down the components further. A top-down divide-and-conquer approach is employed to dissect tasks into their constituent subtasks. An example of this workflow is illustrated above.

Features are deconstructed recursively into subtasks, and solutions are built up from the tasks in the leaf nodes. A significant advantage of this semi-structured approach is the naturally high cohesion in the artefact being produced. Since this approach offers a bird's eye view of the tasks, it is instrumental in promoting the development of reusable code.

Figure 5.1 illustrates how the Parser component is broken down into its subtasks: locating endpoints, retrieving methods, tracing method calls and extending file methods through imports. There are numerous methods that carry out the same functionality across the Java and Python implementations, with the only differentiation being in the file imports (represented by the grey nodes).

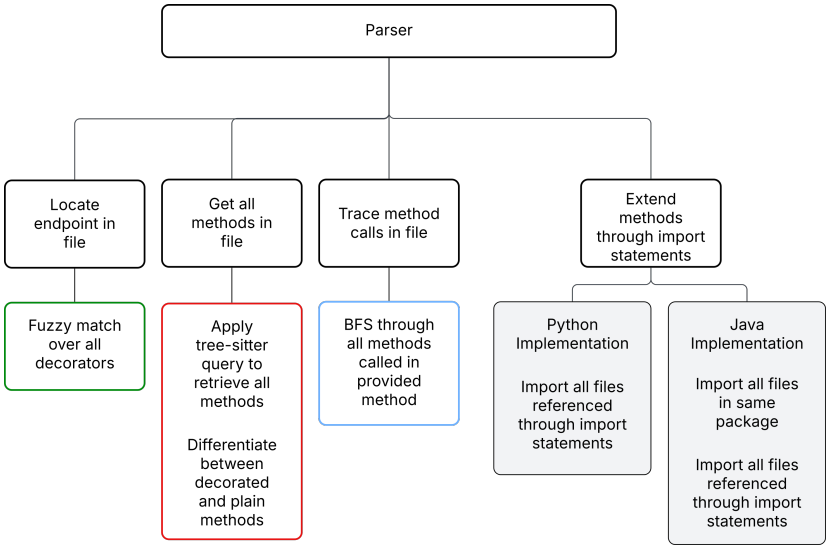

Figure 5.2: Simplified Break Down of Parser Components

By evaluating Figure 5.1, Figure 5.2 can be derived. It is undeniable that a top-down approach offers a clear opportunity to uncover repeated code. The resulting simplification almost halves the development time.

5.3 Hardware

This section provides an overview of the hardware used in developing the tool and training the deep-learning classifier (Table 5.1, Table 5.2).

5.3.1 Tool Development

Hardware Usage	Hardware Specification
Artefact development and testing	Apple MacBook Pro M2 (2023) 16 GB RAM macOS Sequoia
Hosting Ollama server	vast.ai (cloud GPU) A100 SXM4 80 GB RAM AMD EPYC 7532 150 GB disk space Nvidia CUDA + CuDNN Development base image Ubuntu 22.04
Hosting web server test bed and Instana agent	AWS EC2 t2.medium 30 GiB disk space Amazon Linux 2

Table 5.1: Hardware Usage and Specifications

5.3.2 Classifier Fine-Tuning

Hardware Usage	Hardware Specification
Preprocessing training data	Paperspace Compute C9 24 CPU 129 GB RAM 150 GB disk space Ubuntu 22.04
Fine-tuning deep-learning model	Vast.ai (cloud GPU) A100 SXM4 80 GB VRAM GPU AMD EPYC 7532 150 GB disk space Nvidia CUDA + CuDNN Development base image Ubuntu 22.04 Paperspace H100 (Hopper) 16 CPU 268 GB RAM 150 GB disk space ML-in-a-Box base image Ubuntu 22.04

Table 5.2: Hardware used for preprocessing and fine-tuning

5.4 Software

This section provides an overview of the software used in developing the tool (5.3).

Software Usage	Software Tool
Integrated development environment (IDE)	Visual Studio Code Version 1.99.0
Container engine for test bed, Ollama and Instana agent	Docker
Application Performance Monitoring (APM)	IBM Instana
Implementing source code	Python 3.13
Dependency management	Poetry

Table 5.3: Software Tools used Across Development and Deployment Stages

5.4.1 Libraries

A number of Python libraries are adopted for the development of this tool. The following is a brief description of each and their usage.

Prefect

Prefect is a framework for managing and tracking workflows [47]. The pipes and filters design pattern mimics a linear workflow, and this tool enables users to view the progress of flows through an intuitive dashboard.

Typer

Typer is a library that serves to streamline the creation of command-line tools [51].

RapidFuzz

RapidFuzz is a utility that performs approximate string matching [5]. This paper adopts this tool to match Instana's labels for slow endpoints with the decorator labels used in the source code. This utility is also employed to facilitate user convenience. By fuzzy matching on the repository names, users are not required to provide exact names.

PyGitHub

PyGitHub is a wrapper around the GitHub API [57]. This utility is employed to streamline the extraction and insertion of source code.

Tree-sitter

Tree-Sitter is a tool that produces abstract syntax trees from source code [9]. This is used to extract methods and evaluate import statements in Java and Python source code files. Another significant advantage of using tree-sitter is the well-structured query engine. The query engine provides a robust mechanism to efficiently navigate the generated abstract syntax tree [9].

Hugging Face Transformers

Transformers is a library used to train models and perform inference [68]. It is designed for ease of use and handles the underlying complexities involved in maximising GPU usage during training [68]. This paper adopts the transformers library to train the deep-learning classifier and perform inference.

Pandas

Pandas is a tool built on top of numpy that performs operations on structured data [39]. Pandas is used to preprocess the IBM CodeNet dataset to produce labelled data for training.

5.5 Component Implementation

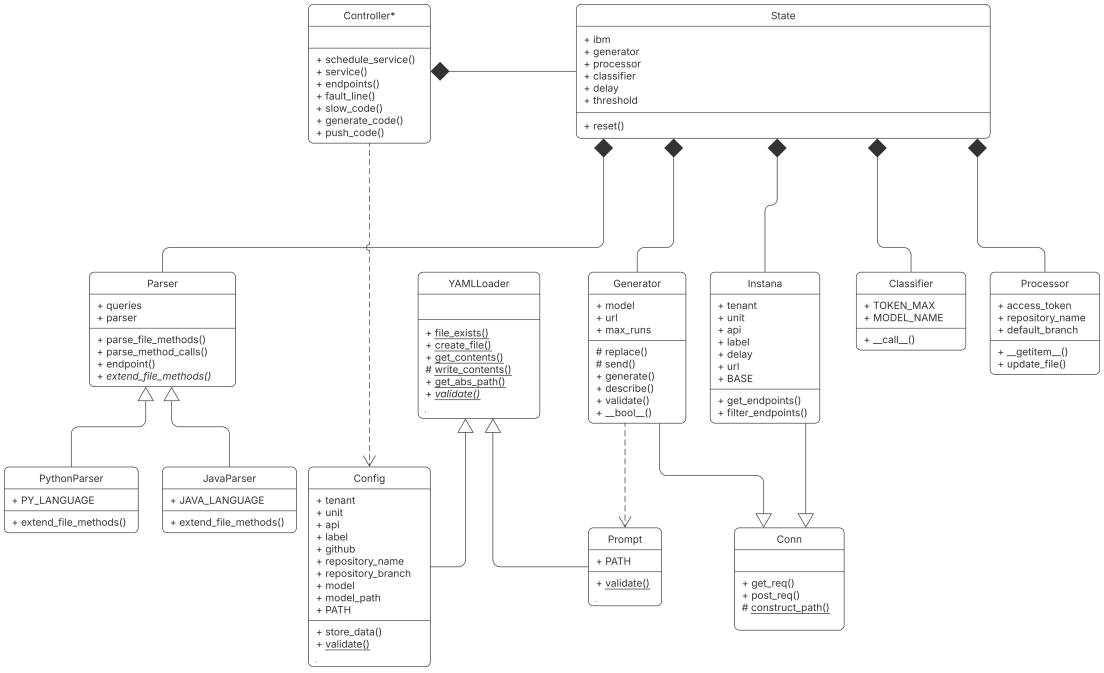

Figure 5.3: UML Class Diagram of the Tool

Figure 5.3 seeks to highlight the relationship between the different components using a UML class diagram. The 'State' class has a composite relationship with each principal component. It is mutated by the 'Controller' class. The diagram does not include the 'MethodNode' and 'FileNode' classes for brevity. These utility classes are used by the 'Parser', 'Generator', 'Processor' and 'Classifier'. Another example of reusable components is the 'Conn' and 'YAMLLoader' classes. These provide the necessary infrastructure to make requests and store configuration files, respectively.

This UML class diagram demonstrates the concrete benefits of using the feature-driven development methodology in tandem with the divide-and-conquer strategy.

It would be useful at this stage to consider the file structure. Figure 5.4 shows an extract of the file system. The tests folder has been collapsed for brevity.

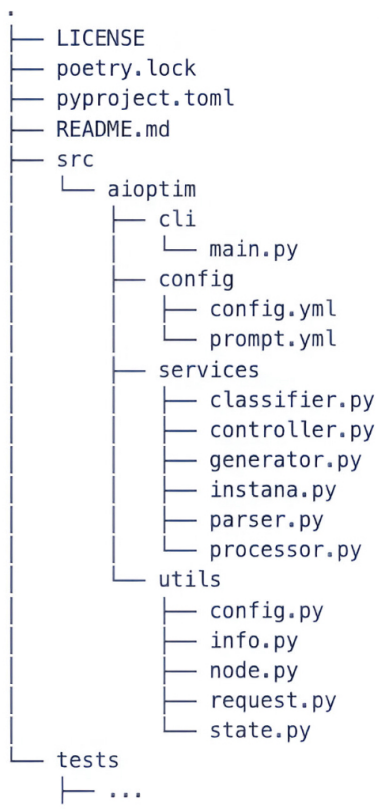

Figure 5.4: File Structure of the Tool

The pipeline components are stored under the 'services' folder, whereas standard utilities are stored under the 'utils' folder. The decision to structure folders in this manner was motivated by the need to offer a modular adaptive approach for developers seeking to extend this tool. A by-product of modularisation is the separation of concerns, which reduces the coupling between components.

5.5.1 Controller

As mentioned in the previous chapter, the implementation revolves around the pipes and filters design pattern. The controller provides the plumbing between the filters, and a shared mutable state object flows through each stage. Further stages can be added by amending the list of filters (1).

Algorithm 1 Scheduled Service

```
filters ← [endpoints, fault_line, slow_code, generate_code, push_code]
state ← initialised State
for all filter in filters do
    filter(state)
end for
```

5.5.2 Instana

Prior to running the tool, it is assumed that IBM Instana has been configured with an application perspective.

Slow endpoints are fetched from the Instana backend using the curl command illustrated in Figure 5.5.

Consequently, the response is filtered to expose endpoints that exceed the user-specified latency thresholds.

```
curl -i -X POST \
  'https://unit-tenant.instana.io/api/application-monitoring/metrics/endpoints?fillTimeSeries=true' \
  -H 'Content-Type: application/json' \
  -H 'authorization: apiToken xxx' \
  -d '{
    "applicationBoundaryScope": "ALL",
    "excludeSynthetic": true,
    "entityType": "HTTP",
    "metrics": [
      {
        "aggregation": "MEAN",
        "metric": "latency"
      }
    ],
    "order": {
      "by": "latency.mean",
      "direction": "DESC"
    },
    "timeFrame": {
      "to": 1744523350860,
      "windowSize": 120000
    }
  }'
```

Figure 5.5: Instana Endpoint Metrics Request

5.5.3 Fault-Line Generator

This component operates as a lightweight parser. It is important to stress that the purpose of this component is to retrieve a complete list of methods that are part of a method trace. To statically extract these methods, the following steps are taken:

1. For each identified slow endpoint, a list of files using the same programming language is pulled from the GitHub repository.
2. For each file that uses the same language, methods are extracted using a tree-sitter query and stored in the file's hashmap.
3. Import statements in each file are evaluated, and the methods in the imported files are merged with those of the importing file.
4. The method block corresponding to the slow endpoint's label is located through a linear search employing fuzzy matching.
5. To obtain the method trace from this method block, a breadth-first search (BFS) is employed, as illustrated in Algorithm 2.

Algorithm 2 Method Trace BFS

```
fault_line ← empty set
queue ← [endpoint_method]
while length(queue) > 0 do
    node ← dequeue(queue)
    fault_line ← fault_line + [node]
    methods ← extract methods using tree-sitter
    for all method in methods do
        if method in file and method ∉ fault_line then
            enqueue(queue, method)
        end if
    end for
end while
```

5.5.4 Classifier

The classifier is trained using labelled Java, Python and JavaScript code samples. The labels for these code samples are derived from the runtime lookup table. This lookup table is derived from the CodeNet metadata. The metadata describes the CPU runtime (milliseconds) for code samples, as Table 5.4 illustrates.

Submission ID	Problem ID	Language	Status	CPU (ms)	Memory (KB)	Code Size (B)
s056752305	p03929	Python	Accepted	27	9020	336
s079240305	p03929	Python	Runtime Error	24	8916	334
s998680993	p03929	Java	Accepted	85	32896	3419
s644811775	p03929	JavaScript	Accepted	71	30108	2478
s670542745	p03929	Python	Accepted	27	9096	343

Table 5.4: CodeNet Metadata

This data is transformed and cleaned into a lookup table using the pandas library. The lookup table groups code samples by the problem they are solving and the language of the implemented solution. The runtime of the code samples within each group is then split into 19 quantiles ranging from 0.05 to 0.95 inclusive. These quantiles indicate the range of runtimes needed for code solutions to be marked as 'fast' or 'slow'. For example, code samples with a runtime that falls below the 0.1 quantile group would be marked as 'fast'. Table 5.5 depicts an excerpt of this data.

Problem ID	Language	0.05	0.1	0.9	0.95
p01916	Java	41.5	43.0	67.0	68.5
p01916	Python	20.0	20.0	30.0	30.0
p03867	Java	110.2	115.4	256.6	264.5
p03867	Python	108.6	121.2	459.0	792.0
p01080	Java	453.5	487.0	1023.0	1056.5

Table 5.5: CPU Time Percentiles

After creating a lookup table, the runtime of each code sample can be evaluated. A caveat to evaluating code samples in such a manner is that samples with runtimes that lie within central quantiles could be subject to measurement inaccuracies, as described in the design chapter. To this end, this paper adopts an approach that utilises the two extremes of the quantile data to label code samples. Two experiments are presented to select the bounds for those extremes.

Experiment A Using 0.25 as a lower quantile bound and 0.75 as the upper quantile bound resulted in an unbalanced dataset of 303268 'fast' code samples and 512036 'slow' code samples.

Experiment B To capture more 'fast' code samples, the experiment was repeated with 0.4 as a lower quantile bound and 0.75 as the upper quantile bound. This resulted in a balanced dataset of 503025 'fast' code samples and 512036 'slow' code samples.

This approach uses bounds close to central quantiles, however a more balanced dataset would make for more effective training. Table 5.6 shows an extract of the resulting labelled data, with the label '0' marking fast code samples and the label '1' marking slow samples.

Problem ID	Language	Submission ID	Label
p03668	Python	s190135922	0
p03668	Python	s459255807	1
p03654	Python	s231407463	0
p03654	Python	s124832613	1
p03668	Java	s697947092	1
p03668	Java	s201931451	1
p03668	Java	s445076481	1
p03668	Java	s012948885	0
p03668	Java	s570167340	0
p03668	Java	s389976734	1
p03668	Java	s746455707	0
p03668	Java	s662099250	1
p03668	Java	s406645243	0
p03668	Java	s699116788	0
p02238	JavaScript	s310070390	0
p02238	JavaScript	s408802505	1
p02238	JavaScript	s063621544	1

Table 5.6: Labelled Submission Training Data

Training

Training the BERT model is then conducted using cloud GPU providers, namely Paperspace and Vast.ai. The following parameters were used:

- Learning rate: 2e-5
- Epochs: 3.0
- Batch size: 64
- Shuffled dataset split: 80% training and 20% evaluation.
- Optimiser: AdamW

Inference

Hugging Face provides a pipeline utility to perform model inference. This streamlines the process of using fine-tuned models for classification. Accurate inference results are conditional on the input sequence adhering to the model's token limits. GraphCodeBERT-base has a token limit of 512.

Therefore, in the implementation of this tool, code that exceeds the model's token limits is broken down into segments and passed through the pipeline separately. The final classification is based on the mode of the segment classifications.

5.5.5 Code Generator

This section will describe the implementation of the code generation component. The code generation component operates on preloaded models served on Ollama. This offers a flexible strategy for users to switch models as they become more effective.

To make use of Ollama, this tool first queries the available models using the curl command below illustrated in Listing 5.1.

Listing 5.1: Ollama Model Retrieval

```
curl http://localhost:11434/api/tags
```

If the user-specified model is found on the Ollama server, code generation requests are made to replace the identified slow methods. Figure 5.6 presents the curl command to generate more performant code.

The prompt is constructed using a base prompt and dynamic data that replaces distinct keys in the base prompt. As discussed in the background, the base prompt is formed using engineering

```
curl http://ollama-server-ip:11434/api/generate -d '{
  "model": "user-provided-model",
  "prompt": "You are an expert at software engineering...",
  "stream": false
}'
```

Figure 5.6: Ollama Generation Request

techniques such as ‘information retrieval’ and ‘context amplification’ [46].

Conversely, an example of dynamic data includes code blocks and method signatures. This dynamic data is merged with a static base prompt through the mechanism of templating. Templating seeks to replace placeholder information with dynamic data. To differentiate placeholder information and text that belongs to the base prompt, boundary characters are employed. Listing 5.2 demonstrates that this tool employs the dollar symbol as the boundary character.

Listing 5.2: Prompt Structure

```
Prompt: Rewrite the code... Code Snippet: $CODE$
```

Turning to the model’s response, it is critical to ensure that the generated code is semantically the same as the code it will replace. Thus, the generated code undergoes validation. A slight modification of the CodeJudge framework is adopted to perform this validation. In a broad sense, this framework queries a model to assert that code meets some predefined requirements. In order to function effectively, requirements must be derived from the original piece of code. To facilitate this step, an additional prompt slots between the CodeJudge framework and the code generation phase.

This strategy is combined using a prompt chaining technique to repeatedly generate code until it conforms to the specification or a set number of tries have passed. These conditions exist to balance the quality of the responses against the costs associated with running GPUs. Manual evaluation suggests that three tries are often enough to generate high-quality code in most cases. It is impossible to provide an exact number of tries that would always result in a high-quality response because of the non-determinism involved with generating responses. Once validation has passed, the original code is replaced by the generated code and pushed onto GitHub.

5.6 Interface

To conform to the requirements, this tool implements an accessible user interface.

The interface is made accessible through the use of:

- A limited instruction set - setup (Figure 5.8) & start (Figure 5.7)
- A guided setup process (Figure 5.10)

- Colour coded output (Figure 5.9)

```
Usage: aioptim setup [OPTIONS]

Sets the parameters for the AI Optimiser tool.
NOTE: Ensure the GitHub PAT is associated with a single repository
NOTE: Ensure the IBM API Key is prefixed with the 'apiToken'

 Options
 --help          Show this message and exit.

 IBM Details
 *   -tenant     TEXT  IBM tenant from https://unit-tenant.instana.io [default: None] [required]
 *   -unit       TEXT  IBM unit from https://unit-tenant.instana.io [default: None] [required]
 *   -api        TEXT  IBM user API Key [default: None] [required]
 *   -label      TEXT  IBM label associated with applications perspective [default: None] [required]

 GitHub Details
 *   -pat        TEXT  GitHub PAT with repository read/write permissions [default: None] [required]
 *   -repo       TEXT  The name of the repository with read/write permissions [default: None] [required]
     -branch     TEXT  The name of the deployment branch [default: main]

 Ollama Details
   -model      TEXT  The Ollama model to use when generating code [default: codellama]
   -ollama     TEXT  The API path for the Ollama model [PATH]:[PORT] [default: http://localhost:11434]
```

Figure 5.7: Setup Command Options

```
Usage: aioptim start [OPTIONS] [THRESHOLD] [DELAY]

Checks the setup parameters and starts the service.

 Running Parameters
   threshold      [THRESHOLD]  The maximum endpoint threshold in millisecond [default: 500]
   delay          [DELAY]      How often (minutes) to run the Instana service [default: 2]

 Options
 --help          Show this message and exit.
```

Figure 5.8: Start Command Options

```
| INFO    | Flow run 'ochre-owl' - Beginning flow run 'ochre-owl' for flow 'entry-point'
| INFO    | Task run 'get-slow-endpoints-a1c' - 500
| INFO    | Task run 'get-slow-endpoints-a1c' - Finished in state Completed()
| INFO    | Task run 'get-fault-line-58e' - Finished in state Completed()
| INFO    | Task run 'get-slow-code-71b' - Finished in state Completed()
| INFO    | Task run 'generate-code-21e' - Finished in state Completed()
| INFO    | Task run 'push-code-1e9' - Finished in state Completed()
| INFO    | Flow run 'ochre-owl' - Finished in state Completed()
```

Figure 5.9: Command Line Interface

```
Input IBM tenant from https://unit-TENANT.instana.io: example-TENANT
Input IBM unit from https://UNIT-tenant.instana.io: example-UNIT
Input IBM API key: example-API
Input the label for the applications perspective: example-LABEL
Input valid Github PAT with read/write permissions: example-PAT
Input the repository with read/write permissions: example-REPOSITORY
Input the name of the deployment branch [main]: example-BRANCH
Input the Ollama model to use to generate code [codellama]: example-MODEL
Input the URL path to the Ollama model [PATH]:[PORT] [http://localhost:11434]: http://example-PATH:11434
Setup completed successfully
```

Figure 5.10: Prompted Setup Process

Additionally, this tool incorporates the Prefect framework. The benefits of this approach are:

- An intuitive dashboard to monitor the execution of this tool
- Access to historical records

5.7 Further Enhancements

Several enhancements have been made to improve the robustness of this tool.

One such example is the adoption of fault tolerance mechanisms. Each component raises and catches exceptions, providing users with material information to remedy errors.

Additionally, the Typer library has been employed to provide a prompt-based user interface. This feature removes the need to input one lengthy command to set up the tool.

Object-oriented programming principles such as abstraction and inheritance have been employed to ensure the code is extensible. The YAMLLoader class in Figure 5.3 illustrates these principles in practice.

Fuzzy matching is used to match user-provided repository names and Instana labels. This ensures programs do not fail due to slight mismatches in the expected string format.

5.8 Challenges and Solutions

When implementing the classifier, code that exceeds the model's max token length is split into segments. More specifically, the code is encoded and split into segments of 512 tokens, and each segment is decoded back into code. When these decoded segments are sent to a model using the Hugging Face pipeline library, the pipeline encodes the input again before passing it through the model. Occasionally, the pipeline encodes the segment to more than 512 tokens, resulting in a message warning about the inaccuracy of the model output. This is an interesting issue that highlights the imperfect encoding and decoding process. The code is split into 450 tokens instead of 512 to limit further warnings.

Another challenging aspect is transferring the Project CodeNet dataset to the cloud GPU systems to train the classifier. Because of the many files in the dataset, the number of available

inodes in the cloud server is exhausted immediately. Inodes are files that store metadata about files on a filesystem. Exhaustion of inodes means that further files cannot be transferred even if disk space is available. To support the dataset, this paper employs the solution of reformatting the drives to support a greater number of inodes.

5.9 Limitations

While the deep learning model is fine-tuned on Python, Java and JavaScript, the parser is limited to optimising Java and Python source code. This is because the flexibility of JavaScript's syntax [4] means that the tree-sitter query has to account for a number of ways to match function declarations. In turn, this makes it unwieldy to include for the purposes of this dissertation. To support further development, the parser class is devised with the open/closed principle in mind.

5.10 Testing

Testing is a fundamental aspect of software development. It is the process of ensuring that source code is functional and meets the stakeholders' requirements.

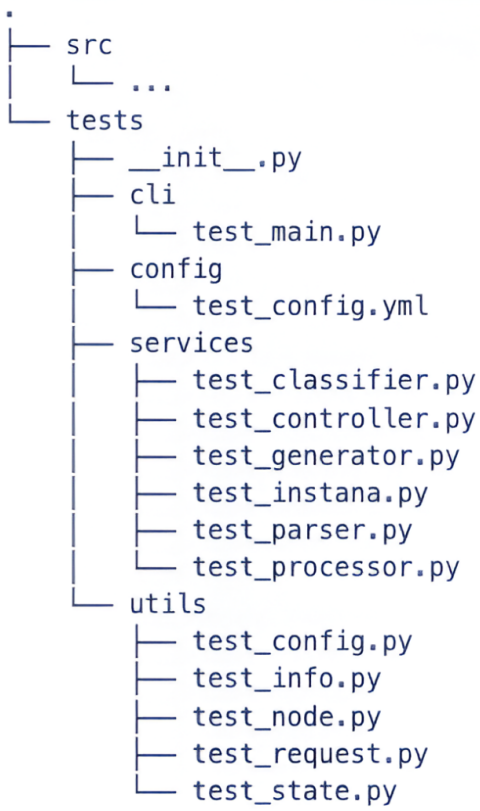

Figure 5.11: File Structure for Tests

File ▲	statements	missing	excluded	coverage
src/aioptim/__init__.py	0	0	0	100%
src/aioptim/cli/__init__.py	0	0	0	100%
src/aioptim/cli/main.py	19	0	2	100%
src/aioptim/services/__init__.py	0	0	0	100%
src/aioptim/services/classifier.py	32	0	0	100%
src/aioptim/services/controller.py	66	0	8	100%
src/aioptim/services/generator.py	32	0	0	100%
src/aioptim/services/instana.py	23	0	0	100%
src/aioptim/services/parser.py	82	0	3	100%
src/aioptim/services/processor.py	40	0	0	100%
src/aioptim/utils/__init__.py	0	0	0	100%
src/aioptim/utils/config.py	93	0	4	100%
src/aioptim/utils/info.py	21	0	0	100%
src/aioptim/utils/node.py	29	0	0	100%
src/aioptim/utils/request.py	15	0	0	100%
src/aioptim/utils/state.py	17	0	0	100%
Total	**469**	**0**	**17**	**100%**

Figure 5.12: Code Coverage Results

There are several different types of testing frameworks, such as unit testing, integration testing, and performance testing. Owing to the structure of our tool, this paper adopts the unit testing strategy. A popular Python framework for unit testing is Pytest. This framework simplifies the process by providing fixtures and support for plugins. In addition, we use the built-in unittest library.

As demonstrated in Figure 5.11, tests are separated into a file structure that mimics the organisation of the source code. A total of 79 unit tests are implemented. They provide 100% code coverage, as illustrated in Figure 5.12.

Chapter 6

Legal, Social, Ethical and Professional Issues

This chapter reviews the various concerns and demonstrates how the development of this tool adhered to best practices. The principles outlined here were paramount to the design and development of this tool.

A significant advantage of adhering to these guidelines is the simplification of the design process. For example, by adopting an accessibility-centred approach, we decided to reduce the parameters involved in setting up. A by-product of this design choice was that the tool became simpler for everyone.

6.1 Legal Issues

Legal frameworks provide scaffolding for the design and implementation of new tools, protecting developers, consumers, and intellectual property holders.

The third-party libraries described in the implementation chapter are open-source and permissive. Furthermore, source code snippets are fairly attributed. The CodeNet dataset is released under the CDLA Permissive 2.0 license. In other words, this license does not impose limits on the artefacts derived from this dataset [35]. To this end, this undertaking adhered to the Copyright, Designs and Patents Act 1988 [65].

Lastly, this tool does not collect any personal data, as defined by Article 4 of the General Data Protection Regulation (GDPR) [17].

6.2 Ethical & Social Considerations

Design decisions were based on accessibility to enable all users to use this tool productively. These decisions include supplementing the command line interface with a graphical web interface to empower users with accessibility needs. When designing this tool, there was a concern with the ever-increasing number of parameters to set the tool up. To limit cognitive overload, a prompt based mechanism was employed.

As this tool has read/write access to repositories, safeguards have been implemented to eliminate any harm to digital property. These include but are not limited to, retry mechanisms to obtain functional code and restrictions on pushing code into the main or deployment branch. These mechanisms were inspired by the OECD AI principles [44]. These principles prescribe the actions necessary for AI tools to integrate with the modern world in a transparent and fair manner [44].

There is growing apprehension that artificial intelligence will replace the IT workforce. It is understandable to be concerned about the many use cases for this tool and related technologies. However, this tool has been designed to augment the capabilities of developers rather than replace them. Developers are given control over the parameters that drive the tool's optimisation decisions. This is in line with the OECD AI Principles about transparency [44].

Lastly, when fine-tuning models, it is imperative to consider the carbon footprint of the training process. To limit our carbon footprint, preprocessing was employed to label the data before being passed to the cloud GPUs.

6.3 Professional Issues

BCS, The Chartered Institute for IT, outlines a code of conduct to set the standards expected of IT professionals [7].

This undertaking:

- Respected the rights of the public
- Conducted due diligence to prevent the utilisation of proprietary code
- Ensured the approach to development was accessibility friendly
- Made justifiable claims with evidence
- Executed actions based on informed decisions.

Additionally, models developed in this project are made available to the public through a permissive Apache 2.0 license. As of April 2025, the main model has over 1000 downloads last month. [1]

Licences

- **Prefect Licence**: `https://github.com/PrefectHQ/prefect/blob/main/LICENSE`
- **Typer Licence**: `https://github.com/fastapi/typer/blob/master/LICENSE`
- **RapidFuzz Licence**: `https://github.com/rapidfuzz/RapidFuzz/blob/main/LICENSE`
- **PyGitHub Licence**: `https://github.com/PyGithub/PyGithub/blob/main/COPYING`
- **Tree-sitter Licence**: `https://github.com/tree-sitter/tree-sitter/blob/master/LICENSE`
- **Transformers Licence**: `https://github.com/huggingface/transformers/blob/main/LICENSE`
- **Pandas Licence**: `https://github.com/pandas-dev/pandas/blob/main/LICENSE`

[1] Hugging Face Model

Chapter 7

Evaluation

7.1 Introduction

This chapter aims to assess the effectiveness of the implemented pipeline through a quantitative and qualitative lens. This evaluation discusses how the core components of this pipeline each contribute to and undermine the process of optimising code.

This chapter

- compares three deep learning models in the task of code classification
- compares three generative AI models in the task of code improvement
- conducts an ablation study on Instana and the deep learning models

This evaluation ties in with the objectives of this dissertation, which are to demonstrate how analysis tools can be augmented with AI to identify and resolve performance bottlenecks.

7.2 Classifier Evaluation

A few experiments were crafted to demonstrate the effectiveness of deep learning in code performance classification.

These experiments compared different model configurations and the impact of these configura-

tions on the model's accuracy, precision, recall and F1 score [26]. The parameters described in the implementation chapter were used to train all three models. The independent variables of this experiment were the models and their training sets.

The implementation chapter placed emphasis on the GraphCodeBERT model's ability to comprehend the structure of code [20]. In these experiments, that hypothesis is tested.

The datasets used to train the models in this experiment also come from the previous experiments in the implementation chapter.

This experiment compares:

- GraphCodeBERT-base with a balanced training set from experiment B
- GraphCodeBERT-base with an unbalanced training set from experiment A
- CodeBERT-base with the balanced training set from experiment B

After training, an unseen section of the original dataset was used to evaluate the performance of these models. This dataset consisted of code samples in Java, Python and JavaScript.

Table 7.1: Classifier Performance

Model	**Accuracy**	**Precision**	**Recall**	**F1 Score**
GraphCodeBERT (Unbalanced)	0.850	0.791	0.808	0.800
GraphCodeBERT (Balanced)	0.820	0.853	0.776	0.813
CodeBERT (Balanced)	0.801	0.836	0.753	0.792

Table 7.1 shows the results of these experiments. Before evaluating the data, it is imperative to understand the most important metric for the purposes of this evaluation. Deep learning models were employed to filter out inefficient code from a set of methods. Therefore, the models are expected to be precise. It is also paramount that the models do not overlook a code sample since that code sample could be at the epicentre of performance degradation. To this end, the F1 score presents the best balance [26]. Upon reviewing the F1 scores, it is no surprise that the GraphCodeBERT-base model with the balanced training dataset outperformed the other models. In this case, the hypothesis holds.

Initially, these results were quite promising, and they provided concrete evidence of meeting the SNFR1 requirement.

Looking at the dataset used to train the models, it consisted of a variety of different sized code

samples. This motivated the next experiment, which tested the classifiers' ability to handle lengthy code samples.

The APPS dataset was selected for its higher average complexity of code tasks [24]. Five hundred of these code tasks were chosen at random and provided to a GPT-4o model. The model generated two types of Python code solutions for each task. One solution made effective use of data structures and algorithms, and the other misused them. A sample of these solutions was verified manually.

These 500 rows of data were fed to the three classifiers to gauge their performance when it came to lengthy code samples. The code samples exceeded the tokeniser's max lengths, which meant that the classifier segmented the code and took the mode of its segment classifications.

Table 7.2: Python Classifier Performance

Model	**Accuracy**	**Precision**	**Recall**	**F1 Score**
GraphCodeBERT (Unbalanced)	0.563	0.551	0.676	0.607
GraphCodeBERT (Balanced)	0.537	0.536	0.552	0.544
CodeBERT (Balanced)	0.535	0.542	0.45	0.542

Table 7.2 shows that the deep learning models performed poorly in classifying longer-form code content. This highlights a stark correlation between the length of code input and the accuracy of the model's output.

Table 7.3: Java Classifier Performance

Model	**Accuracy**	**Precision**	**Recall**	**F1 Score**
GraphCodeBERT (Unbalanced)	0.571	0.570	0.576	0.573
GraphCodeBERT (Balanced)	0.537	0.536	0.552	0.544
CodeBERT (Balanced)	0.522	0.530	0.388	0.448

To ensure this was not a language-specific issue, a new dataset of Java code solutions was created using the same approach, and this experiment was repeated. Table 7.3 illustrates the same issues. This approach does not work optimally for lengthy code blocks.

When attempting to find some reasons for this behaviour, an interesting pattern was encountered. Models tend to react negatively to key terms such as 'sort'. Perhaps this is a result of using a training set from competitive coding contests [49], where sort functions are not often used.

7.3 Generative AI Model Evaluation

This chapter aims to demonstrate generative AI's effectiveness in making code more performant. As part of the evaluation, three models were taken into consideration:

- Qwen2.5 coder:32b-instruct
- CodeLlama:34b-instruct
- Opencoder:8b-instruct

These models were chosen for their availability through Ollama and pass@1 [11] score based on the Eval Plus benchmark [36].

In the task of generating replacement code, this paper expected these models to develop high-quality, efficient code. The resulting code was also expected to be semantically similar to the input.

The datasets from the previous experiment were reused to experiment with these models. These datasets included 500 fast and slow code implementations in Python and 500 in Java. For this experiment, the fast code implementations were ignored.

In this experiment, each model was run from the same host machine with the same prompt and input. The model was asked to generate efficient code that could replace the slow code given to it. This meant the generated code should have been semantically similar to the code it replaced.

The generated code is then sent off to a GPT-4o model for comparison. The GPT-4o model is prompted to verify the syntactic functionality, semantic similarity and performance of the generated code. If the code implementation meets the requirements, a positive signal will be sent back. A sample of results were evaluated to ensure that GPT-4o was behaving as expected. These experiments measured the number of positive signals for Python and Java code for each

Table 7.4: Optimised Code Generation Success Rate

Model	**Python (%)**	**Java (%)**
CodeLLaMA 34B	19.2	16.2
OpenCoder 8B	43.6	48.0
QwenCoder 32B	72.4	73.8

model. As per Table 7.4, qwen-coder outperforms all the models. This demonstrates that there can exist AI agents, which reliably generate performant code.

For the purposes of performance engineering, it is not enough to consider a model's pass@k alone. Another consideration to make was the inference time. This study found that opencoder has the fastest inference times of all three models. This could be important for time-sensitive issues.

Another reflection is that a model's parameter count does not necessarily mean it is more effective. Opencoder, with a parameter count of 8b, outperformed CodeLlama, which had a parameter count of 34b.

Since the budget was a limiting factor, further fine-grained details could not be provided.

7.4 Ablation Study

The following is a qualitative study considering the ablation of the Instana component and the deep learning classifier.

7.4.1 Instana

Instana contributes to the system by flagging the endpoint experiencing severe performance degradation. This greatly simplifies the parser's work, as it only needs to construct a method trace from that endpoint. To simplify, Instana significantly reduces the search space of endpoints.

By removing Instana, the system is forced to use slow static analysis techniques [32] to parse the entire source code. This means more method nodes will end up in the classification queue. A greater number of nodes in the classification queue means a greater chance of false positives, which in turn means a greater likelihood of unnecessary code changes by the generative AI step.

7.4.2 Deep Learning Classifier

The deep learning model filters a list of methods in a trace to pinpoint the possible slow code. By removing the classifier, there is no longer a filter on the outputs of the parser stage. All

the methods detected in a method trace would be sent to the generative AI stages. This also dramatically increases the likelihood of optimising code that is already optimised.

7.5 Reflection

This system has demonstrated to be quite capable of locating performance bottlenecks and automating the resolution process.

During informal testing, the tool was able to detect inefficient code nested quite deep into the test application. In some interesting cases, it improved the functionality of the program in unexpected places. This was quite surprising as it recognised the test application would benefit from certain efficiency measures.

This system could considerably benefit from:

- further training of the classifiers on lengthy code samples
- additional language support (e.g. JavaScript, Go)
- robust mechanisms to prevent occasional empty pushes to the repository

Chapter 8

Conclusion and Future Work

This dissertation explored how dynamic and static analysis tools can harness artificial intelligence to identify and resolve performance bottlenecks automatically.

Generative artificial intelligence is a well-suited approach to generating performant and functional code. This is conditional to the model selected for generation. A model's suitability should not be based on its parameter count. This was a key finding as opencoder (8b) outperformed CodeLlama (34b) in generating performant and functional code.

Binary classification via deep learning is a satisfactory approach to separating code samples based on their performance. This approach is limited to code samples that fit in the tokeniser's max length. Segmenting code samples and performing binary classification across those segments yields slightly better results than guessing.

A hybrid approach to static and dynamic code analysis provides the best of both worlds in terms of speed and accuracy of analysis.

Currently, the tool supports Java and Python frameworks. The parser framework has been designed with extensibility in mind. This would enable developers to add language support to this tool without breaking the pipeline's functionality.

It would be interesting to compare this tool with one that uses only generative artificial intelligence to classify code samples. The CodeJudge framework [64] could be modified to check if certain method blocks contain inefficient code.

Lastly, this tool can form part of a plugin used to evaluate the health of a running application

regularly. This can provide a constant feedback loop to developers, ensuring that their code adheres to the highest programming standards.

References

[1] Michal Aibin and Baeldung. Guide to java.util.concurrent.locks — baeldung, February 2017. Accessed on 2025-01-16.

[2] Amro Al-Said Ahmad, Lamis F. Al-Qora'n, and Ahmad Zayed. Exploring the impact of chaos engineering with various user loads on cloud native applications: an exploratory empirical study. *Computing*, 106(7):2389–2425, Jul 2024.

[3] Abid Ali. A complete guide to data augmentation. https://www.datacamp.com/tutorial/complete-guide-data-augmentation, December 2024. Last updated December 9, 2024. Accessed April 16, 2025.

[4] Gabor Antal, Zoltan Toth, Péter Hegedüs, and Rudolf Ferenc. Enhanced bug prediction in javascript programs with hybrid call-graph based invocation metrics. *Technologies*, 9:3, 12 2020.

[5] Max Bachmann. rapidfuzz/rapidfuzz: Release 3.8.1. https://doi.org/10.5281/zenodo.10938887, April 2024. Version v3.8.1, published on Zenodo.

[6] Ali Basiri, Niosha Behnam, Ruud de Rooij, Lorin Hochstein, Luke Kosewski, Justin Reynolds, and Casey Rosenthal. Chaos engineering. *CoRR*, abs/1702.05843, 2017.

[7] BCS, The Chartered Institute for IT. Bcs code of conduct for members. https://www.bcs.org/media/2211/bcs-code-of-conduct.pdf, 2022. Version 8, reviewed by Trustee Board on 8 June 2022. Accessed: 2024-04-16.

[8] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM*, 53(2):66–75, February 2010.

[9] Max Brunsfeld and Contributors. *Tree-sitter*, 2025. Available at `https://tree-sitter.github.io/tree-sitter`.

[10] Srividhya Chandrasekaran. Enhancing developer experience by reducing cognitive load: A focus on minimization strategies. *International Journal of Computer Trends and Technology*, 72:104–108,, 01 2024.

[11] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *CoRR*, abs/2107.03374, 2021.

[12] I-Hsin Chung, Guojing Cong, David Klepacki, Simone Sbaraglia, Seetharami Seelam, and Hui-Fang Wen. A framework for automated performance bottleneck detection. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–7, April 2008.

[13] Sean Coughlin. Advanced debugging techniques for software engineers, June 2024. Sean Coughlin's Blog. Accessed: 2025-04-16.

[14] Blake Cutler. Firefox & page load speed – part ii, April 2010. Accessed: 2025-04-18.

[15] Matthew Davis and Dylan Theriot. Dynamatic: A race detection tool combining static and dynamic analysis. Undergraduate Research Scholars Thesis, Texas A&M University, May 2021. Submitted to the LAUNCH: Undergraduate Research office in partial fulfillment of the requirements for the designation as an Undergraduate Research Scholar.

[16] Paweł Dymora and Andrzej Paszkiewicz. Performance analysis of selected programming languages in the context of supporting decision-making processes for industry 4.0. *Applied Sciences*, 10, 11 2020.

[17] European Union. General data protection regulation (gdpr) - article 4: Definitions. https://www.legislation.gov.uk/eur/2016/679/article/4, 2016. Accessed: 2024-04-16.

[18] Spandan Garg, Roshanak Zilouchian Moghaddam, Colin B. Clement, Neel Sundaresan, and Chen Wu. Deepdev-perf: a deep learning-based approach for improving software performance. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, page 948–958, New York, NY, USA, 2022. Association for Computing Machinery.

[19] Gaurav Gaur. Pipes and filters pattern: Streamlining data processing in distributed systems, September 2023. Accessed: 2025-04-16.

[20] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. Graphcodebert: Pre-training code representations with data flow. *CoRR*, abs/2009.08366, 2020.

[21] Xue Han and Tingting Yu. An empirical study on performance bugs for highly configurable software systems. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '16, New York, NY, USA, 2016. Association for Computing Machinery.

[22] Robert S. Hanmer. *Pattern-Oriented software architecture for dummies*. John Wiley & Sons, January 2013.

[23] David Harris-Birtill and Rose Harris-Birtill. *Understanding computation time: a critical discussion of time as a computational performance metric*, pages 220–248. The study of time. Brill, Netherlands, September 2021. The 17th triennial conference of the International Society for the Study of Time : Time in Variance ; Conference date: 23-06-2019 Through 29-06-2019.

[24] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring coding challenge competence with apps. *NeurIPS*, 2021.

[25] Kihong Heo, Hakjoo Oh, and Kwangkeun Yi. Machine-learning-guided selectively unsound static analysis. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 519–529, May 2017.

[26] Steven A. Hicks, Inga Strümke, Vajira Thambawita, Malek Hammou, Michael A. Riegler, Pål Halvorsen, and Sravanthi Parasa. On evaluation metrics for medical applications of artificial intelligence. *Scientific Reports*, 12(1), April 2022.

[27] O.S.H. Ho, A. Markiv, I.S.H. Ng, S.J. Han, C. Gasa, A. Muthukumar, V.W.T. Lau, J. Sun, and M.G. Sagoo. Enhancing ai-generated single best answer questions in medical education: A study on the effectiveness of prompt chaining. In *ICERI2024 Proceedings*, 17th annual International Conference of Education, Research and Innovation, page 9761. IATED, 11-13 November, 2024 2024.

[28] Asmaul Hosna, Ethel Merry, Jigmey Gyalmo, Zulfikar Alom, Zeyar Aung, and Mohammad Abdul Azim. Transfer learning: a friendly introduction. *Journal of Big Data*, 9(1):102, Oct 2022.

[29] Cheng Huang, David A. Maltz, Jin Li, and Albert Greenberg. Public dns system and global traffic management. In *2011 Proceedings IEEE INFOCOM*, pages 2615–2623, April 2011.

[30] Olumuyiwa Ibidunmoye, Francisco Hernández-Rodriguez, and Erik Elmroth. Performance anomaly detection and bottleneck identification. *ACM Comput. Surv.*, 48(1), July 2015.

[31] Charlie Lambropoulos. Github, gitlab, bitbucket. which code repository do you use?, October 2022. Accessed: 2025-04-16.

[32] William Landi. Undecidability of static analysis. *ACM Lett. Program. Lang. Syst.*, 1(4):323–337, December 1992.

[33] Anne-Laure Le Cunff, Vincent Giampietro, and Eleanor Dommett. Neurodiversity positively predicts perceived extraneous load in online learning: A quantitative research study. *Education Sciences*, 14(5), 2024.

[34] Shanshan Li, Zhouyang Jia, Yunfeng Li, Xiang-Ke Liao, Erci Xu, Xiaodong Liu, Haochen He, and Long Gao. Detecting performance bottlenecks guided by resource usage. *IEEE Access*, PP:1–1, 08 2019.

[35] Linux Foundation. Community data license agreement - permissive 2.0. `https://cdla.dev/permissive-2-0/`, 2021. Accessed: 2024-04-16.

[36] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation.

In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA, 2023. Curran Associates Inc.

[37] Yang Liu, Weifeng Ma, Xiang Guo, Xuefen Lin, Chennan Wu, and Tianshui Zhu. Impacts of color coding on programming learning in multimedia learning: Moving toward a multimodal methodology. *Frontiers in Psychology*, Volume 12 - 2021, 2021.

[38] Kim Martineau. What is generative ai? https://research.ibm.com/blog/what-is-generative-AI, April 2023. IBM Research Blog.

[39] Wes McKinney. Data structures for statistical computing in python. *Proceedings of the 9th Python in Science Conference*, pages 51–56, 2010.

[40] Neil McKinnon. Pipes & filters - neil mckinnon - medium, October 2018. Medium. Accessed: 2025-04-16.

[41] Jorge A. Navas and Ashish Gehani. Occam-v2: Combining static and dynamic analysis for effective and efficient whole-program specialization. *Commun. ACM*, 66(4):40–47, March 2023.

[42] Anh Nguyen Duc. The impact of software complexity on cost and quality - a comparative analysis between open source and proprietary software. *International Journal of Software Engineering Applications*, 8:17–31, 03 2017.

[43] OpenObserve Team. Tracing python code - module and function call execution, October 2024. Accessed: 2025-04-16.

[44] Organisation for Economic Co-operation and Development. Oecd principles on artificial intelligence. https://oecd.ai/en/ai-principles, 2019. Accessed: 2024-04-16.

[45] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ACL '02, page 311–318, USA, 2002. Association for Computational Linguistics.

[46] Hemil Patel and Shivam Parmar. Prompt engineering for large language model, March 2024. Research Proposal.

[47] Prefect Technologies, Inc. *Prefect*, 2025. Available at https://docs.prefect.io/v3/get-started.

[48] ProductPlan. Feature-driven development (fdd). https://www.productplan.com/glossary/feature-driven-development, nov 2024. Accessed: 2025-04-16.

[49] Ruchir Puri, David Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladmir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, Veronika Thost, Luca Buratti, Saurabh Pujar, Shyam Ramji, Ulrich Finkler, Susan Malaika, and Frederick Reiss. Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks, 2021.

[50] Tarek Ramadan, Tanzima Z. Islam, Chase Phelps, Nathan Pinnow, and Jayaraman J. Thiagarajan. Comparative code structure analysis using deep learning for performance prediction. In *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 151–161, March 2021.

[51] Sebastián Ramírez. *Typer*, 2025. Available at https://typer.tiangolo.com.

[52] Miguel Romero Rosas, Miguel Torres Sanchez, and Rudolf Eigenmann. Should AI Optimize Your Code? A Comparative Study of Classical Optimizing Compilers Versus Current Large Language Models. *arXiv e-prints*, page arXiv:2406.12146, June 2024.

[53] Jordan Samhi, René Just, Tegawendé F. Bissyandé, Michael D. Ernst, and Jacques Klein. Call graph soundness in android static analysis. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2024, page 945–957, New York, NY, USA, 2024. Association for Computing Machinery.

[54] Iqbal H. Sarker. Machine learning: Algorithms, real-world applications and research directions. *SN Computer Science*, 2(3):160, Mar 2021.

[55] Soham Sharma. How to identify and address performance bottlenecks. *Product-Led Alliance*, March 2024. Accessed: 2025-04-18.

[56] R. Sharp and A. Rountev. Interactive exploration of uml sequence diagrams. In *3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 1–6, Sep. 2005.

[57] Benoît Sigoure and Contributors. *PyGithub*, 2025. Available at https://github.com/PyGithub/PyGithub.

[58] Connie U. Smith and Lloyd G. Williams. Software performance antipatterns. In *Proceedings of the 2nd International Workshop on Software and Performance*, WOSP '00, page 127–136, New York, NY, USA, 2000. Association for Computing Machinery.

[59] Hugh Son. Goldman is spending $100 million to shave milliseconds off stock trades. https://www.cnbc.com/2019/08/01/goldman-spending-100-million-to-shave-milliseconds-off-stock-trades.html, August 2019. Published and updated on August 1, 2019. Accessed: 2025-04-18.

[60] Linhai Song and Shan Lu. Performance diagnosis for inefficient loops. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 370–380, May 2017.

[61] Aized Soofi and Arshad Awan. Classification techniques in machine learning: Applications and issues. *Journal of Basic & Applied Sciences*, 13:459–465, 08 2017.

[62] Pengfei Su, Qingsen Wang, Milind Chabbi, and Xu Liu. Pinpointing performance inefficiencies in java. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 818–829, New York, NY, USA, 2019. Association for Computing Machinery.

[63] Tech Talent Charter. Diversity in tech. https://www.techtalentcharter.co.uk/wp-content/uploads/diversity-in-tech-report-2024.pdf, 2024. Slide show.

[64] Weixi Tong and Tianyi Zhang. CodeJudge: Evaluating code generation with large language models. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen, editors, *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 20032–20051, Miami, Florida, USA, November 2024. Association for Computational Linguistics.

[65] UK Government. Copyright, designs and patents act 1988. https://www.legislation.gov.uk/ukpga/1988/48/contents, 1988. Accessed: 2024-04-16.

[66] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[67] Visual Paradigm. What is data flow diagram?, 2025. Accessed: 2025-04-16.

[68] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le

Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45. Association for Computational Linguistics, 2020.

[69] Zhang Yueping, Li Yuefan, and Xu Kesheng. The compound pattern on the chain of responsibility and observer. In *2009 International Forum on Computer Science-Technology and Applications*, volume 3, pages 420–422, Dec 2009.

[70] Saeed Zarinfam. Why Static Analysis Can't Fix Your Performance Problem But Dynamic Analysis Can - Digma, June 2024. Section: Engineering.

[71] Terry Yue Zhuo. ICE-score: Instructing large language models to evaluate code. In Yvette Graham and Matthew Purver, editors, *Findings of the Association for Computational Linguistics: EACL 2024*, pages 2232–2242, St. Julian's, Malta, March 2024. Association for Computational Linguistics.

Chapter 9

Extra Information

9.1 Code with Performance Bottlenecks

Algorithm 3 Dynamic Allocation [58]

for $i \leftarrow 1$ to 10 **do**

$x \leftarrow newObj()$

$x.performAction()$

end for

Algorithm 4 The One-Lane Bridge [1, 58]

procedure SHARED PROCEDURE(a)

$lock.lock()$

try *execute*

finally $lock.unlock()$

end procedure

Algorithm 5 Dead Store [62]

$x \leftarrow 5$

$x \leftarrow 6$

Algorithm 6 Silent Store [62]

$mem[4] \leftarrow 10$

$mem[4] \leftarrow 10$

Algorithm 7 Silent Load [62]

Require: $mem[0] \leftarrow 5$

Require: $mem[1] \leftarrow 5$

$x \leftarrow mem[0]$

$x \leftarrow mem[1]$

Algorithm 8 Resultless loop [60]

for $i \leftarrow 1$ to 10 **do** *nothing*

end for

Algorithm 9 Redundant loop [60]

for $i \leftarrow 1$ to 10 **do**

$x \leftarrow 5 + 5$

end for

Chapter 10

User Guide

10.1 AIOPTIM

Artificial intelligence optimiser (AIOPTIM) is a backend performance optimisation tool. This tool uses static analysis, dynamic analysis, and deep learning to identify endpoints experiencing performance degradation. Subsequently, this tool uses generative artificial intelligence to offer resolutions for the identified endpoints. These changes are pushed back to the repository in a separate branch.

This tool relies on several components, such as IBM's Instana application monitoring tool, an Ollama server and a web server. Below is a guide describing how to run this tool in a macOS 15+ environment. While this tool has been tested on Windows 10+, it is currently unstable.

This tool supports optimising

- Java frameworks such as Spring Boot
- Python frameworks such as Flask and Fast API

This folder contains:

- The tool's source code
- The deep learning model's training code

10.1.1 Hardware

- MacOS (Apple Silicon)
- AWS EC2 (t2.medium, 30GB, Amazon Linux 2)
- Cloud GPU (DigitalOcean, Paperspace and Vast.ai)

10.1.2 Software

- Python 3
- IBM Instana
- GitHub

10.1.3 Preliminary Setup

GitHub

- Fork this repository.
- Create a GitHub Personal Access Token, granting read and write access to the forked repository.

AWS EC2

- Create an AL2 EC2 t2.medium instance with 30GB storage space
- SSH into the EC2 instance
- Install and start the Docker Engine
- Pull the forked repository
- Change directories into the repository and run

```
[] docker-compose up -d
```

- Start the load generation service in the background:

```
[] ./load-gen/load-gen.sh
```

- Retrieve the docker agent command from the 'Deploy Agent' part of the Instana dashboard.
- Inside the EC2 instance, run the docker command to start the Instana agent.

IBM Instana

- Create an application perspective, ensuring the filter is the host/IP of the running EC2 instance.
- If the application perspective does not update after a few minutes, it may be necessary to kill the process running on port 42699 of the EC2 instance.
- Create a personal API token.

Ollama Server

- Create an Ollama server on a cloud GPU platform.
- Ensure that Ollama is listening on 0.0.0.0, as opposed to the loopback address 127.0.0.1
- Ensure that port 11434 on the server is exposed.
- Pull a model into the Ollama server (preferably qwen2.5-coder:32b)
- Ensure that a local curl request can be made to the GPU server.

10.1.4 Using the Tool

In the project's folder, there is a file ending with the 'whl' extension.

Within the same folder:

- Create and source a Python3 environment.
- Run the command below to install the tool.

```
[] pip install aioptim-1.0.0-py3-none-any.whl
```

Upon installation of the tool, the following commands become available:

```
[] aioptim --help aioptim start --help aiotpim setup --help
```

Setup

To set up the tool, run the following command:

```
[] aioptim setup
```

Start

Before starting the tool, on a new terminal window, the following command can be run to start a localhost visualisation dashboard on port 4200:

```
[] prefect server start
```

To start the tool, run this command with parameters for the :

- Threshold: upper bound of acceptable endpoint latency (milliseconds)
- Delay: how often to run the process (minutes)

```
[] aioptim start <threshold> <delay>`
```

This begins the process of locating and resolving slow endpoints. After some time the repository will update with new branches indicating the changes made by the generative models.

10.1.5 Testing

Within the folder, there is a subdirectory titled 'ToolSource'. This file contains the source code for the tool. The following instruction can be run within the repository to run unit tests:

```
[] pytest
```

Chapter 11

Source Code

11.1 Instructions

I verify that I am the sole author of the programs contained in this folder, except where explicitly stated to the contrary. Lavish Kamal Kumar 18/04/2025