Tribhuvan University

Institute of Science and Technology

Central Department of Computer Science and Information Technology

Kirtipur, Kathmandu

A Thesis Proposal On

Comparative Analysis of RoBERTA and CodeBERT for Automated Misconfiguration Detection in Dockerfiles

Under the supervision of Asst. Prof. Bikash Balami

In partial fulfillment of the requirement for Master's degree in Information Technology (MIT), 4th Semester

Submitted to:

Central Department of Computer Science and Information Technology, Tribhuvan University, Kirtipur, Kathmandu, Nepal

Submitted By:

Deepak Aryal (7925005)

(5-2-1181-43-2012)

September, 2025

Table of Contents

List of Figures

List of Abbreviations

AI	Artificial Intelligence
AUC-ROC	Area Under the Receiver Operating Characteristic Curve
BERT	Bi-directional Encoder Representations from Transformers
DSL	Domain Specific Language
ICSC	Indian Certificate of Secondary Education
LSTM	Long Short Term Memory
ML	Machine Learning
NLP	Natural Language Processing
RNN	Recurrent Neural Network
RoBERTa	Robustly Optimized BERT
YAML	Yet Another Markup Language

Chapter 1: Introduction

1.1 Introduction

With the rapid and continuous growth of software, increasing attention is being directed toward infrastructure that enables developers to quickly test, deploy, and scale applications. DevOps tools play a central role in this process by streamlining software delivery and operational efficiency. Among these, Docker has become one of the most widely adopted tools, allowing engineers to build, test, and deploy applications with ease.

The DevOps ecosystem, however, is highly diverse and constantly evolving. As software systems grow in scale and complexity, DevOps tools and their associated artifacts also become more complex. These artifacts often employ specialized input formats; for example, Docker relies on a custom domain-specific language (DSL) to define its configurations. Despite their importance, such DevOps artifacts have historically received limited attention from both academia and industry. Because they are not considered "traditional" code, they frequently fall outside the scope of automated mining, analysis, and research. Developers, therefore, often acquire only task-specific knowledge, with little motivation to develop deeper expertise. As Phillips et al. observe, "if you are good, no one ever knows about it [1]."

Although supportive tools exist such as the VS Code Docker extension, which has millions of installations, they have not guaranteed compliance with best practices. For instance, Dockerfiles available on GitHub still contain significantly more rule violations than those created by experts, leading to misconfigurations that can introduce vulnerabilities and security risks.

This study seeks to address these challenges by conducting a comprehensive evaluation of Natural Language Processing (NLP) approaches for automated misconfiguration detection in Dockerfiles, a critical DevOps artifact in containerized software development. The evaluation will compare models across key performance metrics, including detection accuracy, precision, recall, F1-score, and computational efficiency. In addition, the study will investigate the ability of these approaches to detect previously unseen or novel misconfigurations, which is crucial for ensuring practical reliability and adaptability in real-world scenarios.

By systematically comparing multiple NLP approaches, this research aims to highlight their relative strengths and limitations. The findings will contribute to advancing automated

misconfiguration detection in Dockerfiles and support the development of more reliable tools for securing and optimizing containerized environments. Ultimately, this work seeks to reduce configuration-related vulnerabilities and strengthen trust in modern DevOps practices.

1.2 Problem Statement

In today's digital era, following best practices in tools like Docker is essential, yet many software engineers remain unaware of them. This lack of adherence often results in misconfigurations that expose systems to vulnerabilities and attacks.

Most current solutions, such as Hadolint, Trivy, Dockle, and Chekov, are rule-based and rely on static analysis with predefined security rules. While useful, they do not leverage machine learning or NLP, and existing NLP applications to Dockerfiles have been limited and focused on other tasks.

With the rise of AI, there is growing interest in AI-driven detection methods for Dockerfiles. Unlike static rule-based linters, AI approaches can learn from data, adapt to unseen misconfigurations, and reduce false positives and negatives making them more suitable for real-world DevOps workflows.

This highlights the need for robust automated detection techniques i.e. roBERTa or CodeBERT, that can reliably identify a wide range of Dockerfile misconfigurations, ultimately improving security and trust in containerized systems.

1.3 Objectives

The main objectives of thesis includes;

- To evaluate and compare the performance of RoBERTa and CodeBERT for automated misconfiguration detection in Dockerfiles, using metrics such as accuracy, precision, recall, F1-score, and AUC-ROC.
- To design and develop automated detection framework using either roBERTA or CodeBERT, and find which one has better performance.

Chapter 2: Background Study and Literature Review

2.1 Background Study

Dockerfiles are a type of DevOps artifact used to automate the creation of containerized applications. They specify the base image, dependencies, environment variables, and commands needed to build a Docker image. Misconfigurations in Dockerfiles can lead to failed builds, security vulnerabilities, inefficient images, or runtime errors. With the growing adoption of containerization and microservices, ensuring the correctness and security of Dockerfiles is essential for reliable software deployment.

Common Dockerfile misconfigurations include:

- **Command errors:** Missing or incorrect flags in commands, such as forgetting -y in apt-get install, which can cause builds to hang.
- **Inefficient layering:** Poor instruction ordering that increases image size and build time.
- **Security violations:** Exposing secrets or using outdated base images.
- **Deviation from best practices:** Omitting recommended flags or failing to clean temporary files.

Rule-based tools like Hadolint, Trivy, Dockle, etc are widely used to detect misconfigurations and enforce best practices. While effective for known patterns, they rely on static rules, cannot adapt to unseen issues, and may produce false positives or negatives.

Recent research has explored AI and NLP for analyzing code and Infrastructure-as-Code artifacts. Shipwright applied BERT-style models to detect broken Dockerfiles[2], while FLAKIDOCK used LLMs to repair flaky builds[3]. The Binnacle Project produced a Dockerfile corpus mainly for evaluating linters[4]. In related domains, NLP has been applied to detect misconfigurations in Terraform, Ansible, and Kubernetes YAML files, showing the potential of data-driven approaches to generalize beyond static rules.

Despite the availability of rule-based linting tools, there is very limited research on applying NLP specifically for line-level misconfiguration detection in Dockerfiles. Existing NLP approaches have focused on build failures or repair suggestions, leaving a gap in automated, intelligent detection of subtle or context-dependent misconfigurations. This motivates the present study to explore AI-driven techniques that can complement or surpass traditional tools,

enabling more reliable and adaptive analysis of Dockerfiles in real-world DevOps environments.

2.2.1 Transformers

The Transformer is one of the most important natural language processing (NLP) techniques. It was designed to outperform standard models like long short-term memory networks (LSTMs) and recurrent neural networks (RNNs) when dealing with sequential input, such as text. The Transformer does not do step-by-step word analysis like these models. Instead, it employs a mechanism known as attention, which allows the model to concentrate on the most relevant elements of the text at the same time. This allows it to comprehend the meaning of words in context faster and more precisely.

A Transformer is mainly built from two components: the encoder and the decoder. The encoder reads and represents the input text, while the decoder produces the output. In many tasks like text classification, only the encoder is used. In certain tasks such as translation, the encoder and decoder collaborate.

The core idea behind the Transformer is the self-attention mechanism. Self-attention compares each word with every other word in the input sentence, so the model can learn how words depend on each other, even if they are far apart. This is particularly important when analyzing technical files such as Dockerfiles, where the meaning of one command may depend on another line in the file.

Another important feature of Transformers is that they can be trained in parallel, unlike RNNs and LSTMs which process text one step at a time. Transformers are substantially faster to train on huge datasets, which is why they have formed the foundation for many complex NLP models.

Transformers also use positional encoding to keep track of the order of words, since attention alone does not capture sequence information. This allows the model to understand not only the relationships between words but also their correct order in a sentence.

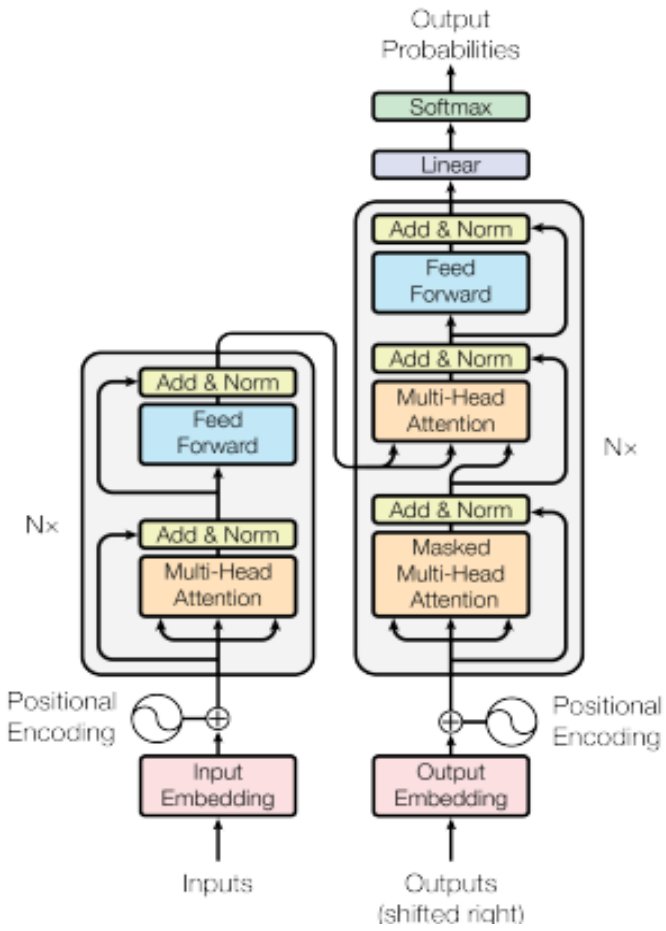

Figure 2.1: Transformer Architecture [5]

2.2 Literature Review

Before beginning this study, several relevant papers were reviewed to gain a deeper understanding of the domain. This section summarizes key insights and findings from the existing literature.

The paper "Learning from, Understanding, and Supporting DevOps Artifacts for Docker" investigates the quality of Dockerfiles authored by experts and non-experts. The study finds that non-expert Dockerfiles violate best practices nearly six times more than expert-authored files, and industrial Dockerfiles show similar issues. The authors develop the Binnacle tool to mine and enforce local rules for improving Dockerfile quality and suggest integrating such tools into IDEs for real-time developer assistance. They highlight that automated, adaptive approaches are needed to address more complex or context-dependent misconfigurations[6].

The paper "Shipwright: A Human-in-the-Loop System for Dockerfile Repair" proposes a system that uses a modified BERT model to embed build logs and cluster broken Dockerfiles. The approach enables automated repair suggestions for faulty Dockerfiles and demonstrates practical improvements in build correctness. While effective for repair, the study primarily focuses on build failure resolution rather than line-level security misconfiguration detection[2].

The paper "Dockerfile Flakiness: Characterization and Repair" introduces FlakiDock, a tool that leverages large language models along with retrieval-augmented generation and dynamic analysis to automatically repair flaky Dockerfiles. The study characterizes common flakiness patterns and demonstrates that FlakiDock significantly improves repair success compared to existing tools. While focused on build flakiness, the approach shows the potential of AI-based techniques for automated Dockerfile correction, although line-level security misconfiguration detection is not addressed[3].

The article "Dockerfile Linting: Every time, Everywhere" discusses the importance of integrating linting tools like Hadolint into Docker workflows to ensure adherence to best practices. It emphasizes the benefits of incorporating linting into the build process to maintain consistent quality across development environments[7].

The paper "Not all Dockerfile Smells are the Same: An Empirical Evaluation of Hadolint Writing Practices by Experts" examines how expert developers perceive and handle Dockerfile smells detected by Hadolint. The study analyzed over 39,000 Dockerfiles and surveyed 37 experts, revealing that even experienced developers sometimes overlook certain smells. The research highlights the importance of linting tools for surfacing Dockerfile issues and informs potential improvements in automated detection, although it does not explore AI or NLP-based misconfiguration detection [8].

The paper "DRIVE: Dockerfile Rule Mining and Violation Detection" proposes an approach to automatically mine implicit rules from Dockerfiles and detect potential violations [9]. The study identifies 34 semantic and 19 syntactic rules, including 9 new semantic rules not previously reported. The findings demonstrate the effectiveness of automated rule mining in improving Dockerfile quality, though it focuses on static analysis rather than AI or NLP-based misconfiguration detection.

Chapter 3: Methodology

The following diagram shows the overall system flowchart.

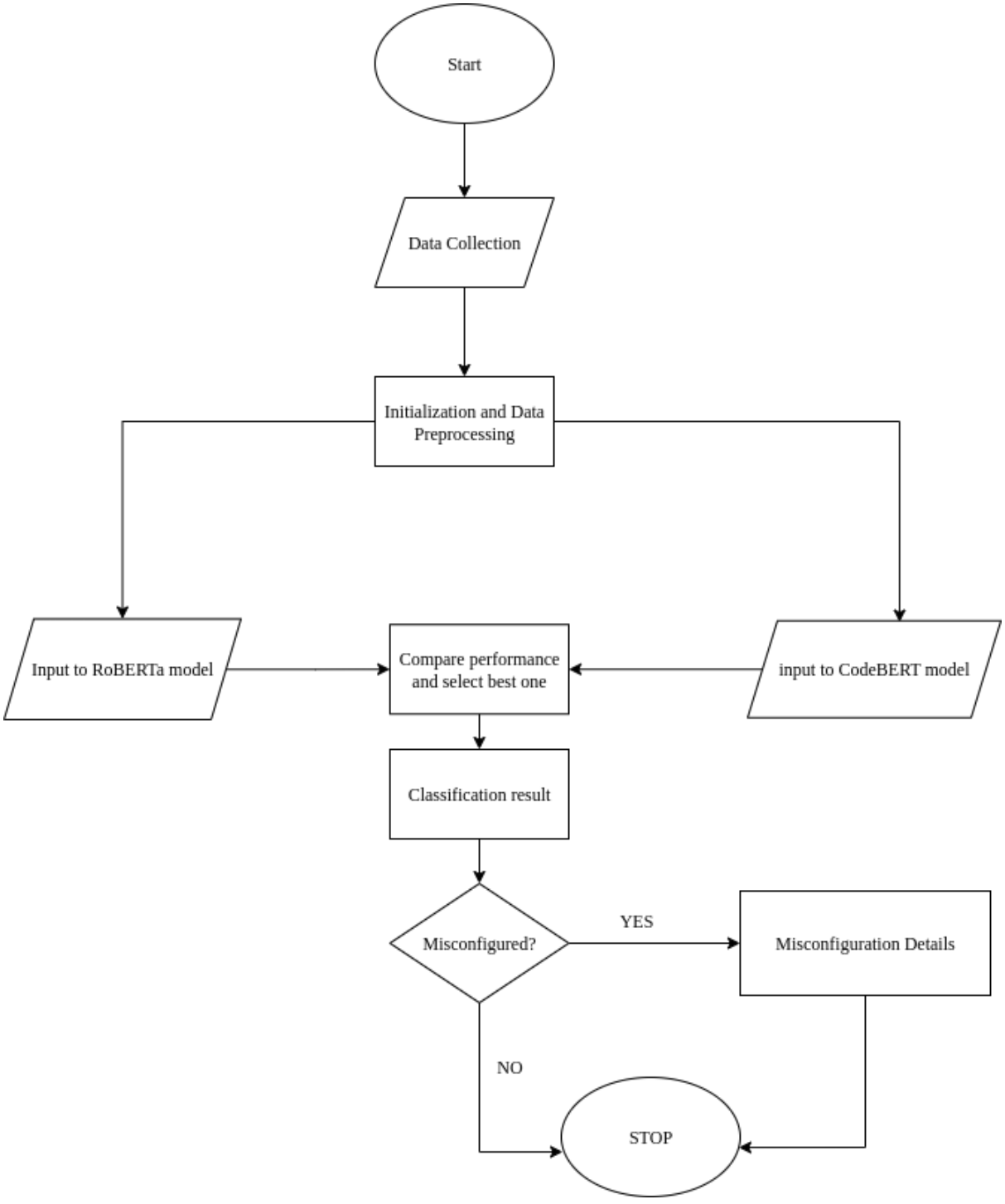

Figure 3.1: System Flow Chart

This flowchart outlines a system that uses a Natural Language Processing (NLP) model to identify misconfigurations. The process begins with Data collection, followed by Initialization

and data preprocessing to prepare the data for the model. The prepared data is then fed as Input to the NLP model for analysis. The model produces a Classification result which is then checked to see if a misconfiguration was detected. If it was, the system generates Misconfiguration details; if not, the process ends. In either case, the system then stops.

3.1 Dataset Description

The dataset for this research will be constructed from publicly available Dockerfiles. The primary source is Henkel et al.'s Dockerfile corpus, which contains approximately 178,000 unique Dockerfiles mined from GitHub repositories[4]. This corpus was introduced in ICSE 2020 and represents real-world Docker usage practices up to around 2019. It has been widely used in empirical studies on software configuration practices and provides a strong foundation for analyzing Dockerfile quality.

However, since Dockerfile syntax and best practices have evolved significantly after 2020 (e.g., changes in artifact handling, multi-stage builds, and security guidelines), relying solely on Henkel's corpus would not reflect current practices. To address this, additional Dockerfiles will be collected using the GitHub Search API, focusing on repositories updated between 2021 and 2025. This ensures that the dataset includes both historical and recent Dockerfile examples, covering the evolution of Dockerfile authoring practices.

Once collected, all Dockerfiles will be parsed line by line, and each command (e.g., FROM, RUN, COPY, WORKDIR) will be extracted for analysis. Each instruction will then be labeled as either correctly configured or misconfigured, forming the basis of the training and evaluation data for this study.

The dataset will be split into 70% training and 30% testing, ensuring a balanced distribution of correct and incorrect configurations across both subsets.

3.2 Implementation Model

3.2.1 RoBERTa

RoBERTa (Robustly Optimized BERT Approach) is an encoder-only Transformer-based model designed for natural language understanding tasks. It is an optimized variant of the original BERT model, trained with larger datasets and improved training strategies, which enhances its ability to capture contextual and semantic information from text.

The architecture leverages two complementary features:

- **Transformer Encoder**: A stack of self-attention layers that produces contextualized representations for each token in the input sequence, allowing the model to understand complex dependencies between words or code tokens. Given an input sequence of tokens $X=\{x_1,x_2,\ldots,x_n\}$, each token is mapped to an embedding vector. The encoder applies self-attention as:

$$Attention(Q,K,V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

where Q, K, V are the query, key, and value matrices derived from the embeddings, and d_k is the dimension of the key vectors. This allows the model to weigh relationships between tokens across the entire sequence.

- **Classification Head**: After encoding, the [CLS] token embedding h_{CLS} is passed through a feed-forward layer to predict the class label:

$$y = softmax(W \cdot h_{CLS} + b)$$

where W and b are trainable parameters, and y represents the probability distribution over classes (e.g., Correct vs Misconfigured).

This combination enables RoBERTa to not only detect whether a Dockerfile contains misconfigurations but also, when fine-tuned appropriately, classify the type of misconfiguration based on learned patterns in the code-like structure of Dockerfile commands

3.2.1 CodeBERT

CodeBERT is a bimodal Transformer-based model designed for both natural language and programming language understanding tasks. It extends the BERT architecture to handle code-related inputs, making it suitable for tasks such as code search, code summarization, and defect or misconfiguration detection in code-like structures.

The architecture leverages two complementary features:

- **Transformer Encoder**: Similar to RoBERTa, CodeBERT uses stacked self-attention layers, but its input consists of both code tokens and natural language tokens. Given an input sequence $X=\{x_1,x_2,\ldots,x_n\}$ (where tokens may represent code or text), the encoder produces contextual embeddings $H=\{h_1,h_2,\ldots,h_n\}$. Each embedding captures semantic relationships across modalities (code + text).

- **Classification or Generation Head**:
  - For classification tasks (e.g., misconfiguration detection), the [CLS] embedding is mapped as:

$$y = softmax(W_c \cdot h_{CLS} + h_c)$$

  - For generation tasks (e.g., code completion), the probability of the next token x_1 is computed as:

$$P(x_t|x_{>t}) = Softmax(W_g.h_t + b_g)$$

where W_c, W_g and b_c, b_g are trainable parameters.

This design enables CodeBERT to understand both the structure and meaning of code, making it well-suited for analyzing Dockerfiles. It can detect misconfigurations by identifying unusual command patterns and classify the type of misconfiguration based on semantic and syntactic relationships within the file.

3.3 Evaluation Metrices:

To evaluate the effectiveness of the proposed models in detecting Dockerfile misconfigurations, both binary classification metrics (correct vs. incorrect) and multi-class classification metrics (categorizing the type of misconfiguration) will be employed.

3.3.1 Accuracy

Accuracy measures the overall correctness of the model's predictions, defined as the ratio of correctly classified instances to the total number of instances:

$$\text{Accuracy}=\frac{TP+TN}{TP+TN+FP+FN}$$

where TP = true positives, TN = true negatives, FP = false positives, and FN = false negatives.

3.3.2 Precision

It is a metric that indicates the percentage of positive predictions that are actually positive, also known as the positive predictive value. It is given by;

$$\text{Precision}=\frac{TP}{\text{TP+FP}}$$

This metric is particularly important when identifying incorrect configurations, as it reflects the reliability of error detection.

3.3.3 Recall

It is a metric that measures how well a model identifies true positives. It is given by;

$$\text{Recall} = \frac{TP}{\text{TP}+\text{FN}}$$

High recall ensures that most misconfigurations are correctly detected.

3.3.4 F1 Score

It is a weighted average of precision and recall, and is calculated using the harmonic mean of these two matrices.

$$\text{F1 Score} = 2 \times \frac{Precision \times Recall}{Precision + Recall}$$

3.3.5 AUC-ROC

Plotting the True Positive Rate (Recall) versus the False Positive Rate at different threshold values is done via the ROC curve. The model's capacity to differentiate between classes is shown by the AUC score, which is a measure of separability.

- AUC = 1 → perfect classification.
- AUC = 0.5 → random guessing.

Better model performance in distinguishing between fabricated and real pictures is indicated by a higher AUC score.

Chapter 4: Expected Outcomes and Work Schedule

4.1 Expected Outcomes

The expected outcome of this dissertation is to determine which NLP model i.e. either RoBERTa or CodeBERT will perform better in detecting misconfigurations in Dockerfiles. The study will compare their performance metrics and computational efficiency. Additionally, the best-performing model will be implemented to not only classify whether a configuration is correct or incorrect, but also to identify and explain the type of misconfiguration. This will contribute to building a more reliable, efficient, and explainable system for automated Dockerfile quality assurance.

4.2 Work Schedule

The project started on 3rd August 2025 and is targeted to be completed within 12 weeks, by the end of October 2025. The Gantt Chart below depicts the working Schedule.

S. N	Activity/Month	1^{st}-2^{nd} week	3^{rd}- 4^{th} Week	5^{th}-6^{th} week	7^{th}-8^{th} week	9^{th}-10^{th} week	11^{th}-12^{th} week
1	Preliminary Study						
2	Literature Review						
3	Data Collection						
4	Model Testing and Analysis						
5	Documentation						

Figure 4.1: Working Schedule

References

[1] S. Phillips, T. Zimmermann, and C. Bird, "Understanding and Improving Software Build Teams," *Proc. - Int. Conf. Softw. Eng.*, May 2014, doi: 10.1145/2568225.2568274.

[2] J. Henkel, D. Silva, L. Teixeira, M. d'Amorim, and T. Reps, "Shipwright: A Human-in-the-Loop System for Dockerfile Repair," *2021 IEEE/ACM 43rd Int. Conf. Softw. Eng.*, pp. 1148–1160, 2021, [Online]. Available: https://api.semanticscholar.org/CorpusID:232045443

[3] T. Shabani, N. Nashid, P. Alian, and A. Mesbah, "Dockerfile Flakiness: Characterization and Repair," *2025 IEEE/ACM 47th Int. Conf. Softw. Eng.*, pp. 1793–1805, 2024, [Online]. Available: https://api.semanticscholar.org/CorpusID:271855816

[4] J. Henkel, C. Bird, S. K. Lahiri, and T. Reps, "ICSE 2020 Artifact for: Learning from, Understanding, and Supporting DevOps Artifacts for Docker." Zenodo, 2020. doi: 10.5281/zenodo.3628771.

[5] A. Vaswani *et al.*, "Attention is All you Need," in *Neural Information Processing Systems*, 2017. [Online]. Available: https://api.semanticscholar.org/CorpusID:13756489

[6] J. Henkel, C. Bird, S. K. Lahiri, and T. Reps, "Learning from, Understanding, and Supporting DevOps Artifacts for Docker," *2020 IEEE/ACM 42nd Int. Conf. Softw. Eng.*, pp. 38–49, 2020, [Online]. Available: https://api.semanticscholar.org/CorpusID:211069127

[7] D. W. Elliott, "Dockerfile Linting: Every time, Everywhere," 2020, [Online]. Available: https://medium.com/@david.w.elliott/dockerfile-linting-every-time-everywhere-d8d271a1e650

[8] G. Rosa, S. Scalabrino, G. Robles, and R. Oliveto, *Not all Dockerfile Smells are the Same: An Empirical Evaluation of Hadolint Writing Practices by Experts*. 2024. doi: 10.1145/3643991.3644905.

[9] Y. Zhou, W. Zhan, Z. Li, T. Han, T. Chen, and H. C. Gall, "DRIVE: Dockerfile Rule Mining and Violation Detection," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, pp. 1–23, 2022, [Online]. Available: https://api.semanticscholar.org/CorpusID:254564701