TRIBHUVAN UNIVERSITY

Institute of Science and Technology

A Thesis Report Submission in

partial fulfillment of the requirement for the Master's degree in Information Technology

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

Under the Supervision of

Bikash Balami

Submitted by:

Deepak Aryal
TU Roll number (7925005)
Reg No. 5-2-1181-43-2012

Submitted to:

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

December, 2025

Tribhuvan University
Institute of Science and Technology

SUPERVISOR RECOMMENDATION

This is to certify that Mr. Deepak Aryal has submitted the thesis report on the topic **"Comparative Analysis of RoBERTA and CodeBERT for Automated Misconfiguration Detection in Dockerfiles**” for the partial fulfilment of Masters of Information Technology, Fourth semester. I hereby declare that this thesis report has been approved.

____

Supervisor

Assistant Professor Bikash Balami

Central Department of Computer Science and Information Technology

Tribhuvan University
Institute of Science and Technology

APPROVAL LETTER

This is to certify that the Thesis report prepared by Mr. Deepak Aryal on the topic "**Comparative Analysis of RoBERTA and CodeBERT for Automated Misconfiguration Detection in Dockerfiles**" in partial fulfilment of the requirements for the degree of Masters of information Technology has been well studied. In our opinion, it is satisfactory in the scope and quality as a project for the required degree.

Evaluation Committee

………………………………………

Asst. Prof. Sarbin Sayami
(H.O.D)
Central Department of Computer Science
and Information Technology

…………………………………………....

Asst. Prof. Bikash Balami
(Supervisor)
Central Department of Computer Science
and Information Technology

…………………………………………

(Internal Examiner)

……………………………

(External Examiner)

ACKNOWLEDGEMENT

The success and final outcome of this report required a lot of guidance and assistance from many people and I am very fortunate to have got this all along the completion. I am very glad to express my deepest sense of gratitude and sincere thanks to my highly respected and esteemed supervisor **Asst. Prof. Bikash Balami,** Central Department of Computer Science and Information Technology for his valuable supervision, guidance, encouragement, and support for completing this paper.

I am also thankful to **Asst. Prof. Sarbin Sayami**, HOD of Central Department of Computer Science and Information Technology for his constant support throughout the period. Furthermore, with immense pleasure, I submit by deepest gratitude to the Central Department of Computer Science and Information Technology, Tribhuvan University, and all the faculty members of CDCSIT for providing the platform to explore the knowledge of interest. At the end I would like to express my sincere thanks to all my friends and others who helped me directly or indirectly.

Thanking you,

Deepak Aryal (7925005)

ABSTRACT

This thesis addresses the problem of automated misconfiguration detection in Dockerfiles using transformer-based Natural Language Processing (NLP) models. Dockerfiles are essential DevOps artifacts, yet they frequently contain configuration errors that may lead to security vulnerabilities and deployment failures. Traditional rule-based analysis tools rely on predefined rules and often fail to generalize to unseen or complex misconfiguration patterns. To overcome these limitations, this research evaluates two pretrained transformer models, CodeBERT and RoBERTa, for intelligent Dockerfile analysis.

A two-stage learning framework is proposed. In the first stage, the models perform binary classification to distinguish correctly configured Dockerfile commands from misconfigured ones. In the second stage, misconfigured commands are classified into specific violation rules, enabling fine-grained misconfiguration detection. The models are trained and evaluated on a large-scale dataset comprising approximately 178,000 Dockerfiles and 1.6 million extracted commands collected from public repositories.

Experimental results show that both models achieve high performance in binary misconfiguration detection, with an accuracy of 98% and ROC–AUC values above 0.99. For rule-specific classification across 63 misconfiguration rules, CodeBERT outperforms RoBERTa, achieving an accuracy of 70.70% and a macro-averaged F1-score of 71.36%, while RoBERTa attains comparable results. The findings demonstrate that transformer-based models are effective and scalable solutions for automated Dockerfile misconfiguration detection and can complement or surpass traditional rule-based approaches in modern DevOps environments.

Keywords: *Dockerfile, Misconfiguration Detection, Natural Language Processing, Transformer Models, CodeBERT, RoBERTa, DevOps, Infrastructure as Code*

TABLE OF CONTENTS

LIST OF ABBREVIATIONS

AI	Artificial Intelligence
API	Application Programming Interface
AUC-ROC	Area Under the Receiver Operating Characteristic Curve
BERT	Bi-directional Encoder Representations from Transformers
CI /CD	Continuous Integration / Continuous Deployment
CSV	Comma Separated Values
DevOps	Development and Operations
DSL	Domain Specific Language
GPU	Graphics Processing Unit
IaC	Infrastructure as Code
ICSC	Indian Certificate of Secondary Education
IDE	Integrated Development Environment
ML	Machine Learning
NLP	Natural Language Processing
RoBERTa	Robustly Optimized BERT
YAML	Yet Another Markup Language

LIST OF FIGURES

LIST OF TABLES

UNIT 1: INTRODUCTION

1.1 Introduction

The rapid expansion of modern software systems has significantly increased the demand for infrastructure that enables fast development, testing, deployment, and scaling of applications. In this context, DevOps practices have become essential, as they bridge the gap between software development and operations while improving efficiency and reliability. Among the many tools supporting DevOps workflows, Docker has emerged as one of the most widely adopted technologies, allowing developers to package applications and their dependencies into lightweight, portable containers.

Despite its advantages, the DevOps ecosystem is highly dynamic and continuously evolving. As applications grow in size and complexity, the tools and artifacts used to manage them also become more sophisticated. Dockerfiles, which define the configuration and build process of container images, are written using a domain-specific language (DSL). Although these artifacts play a crucial role in application security and performance, they have historically received limited attention from both researchers and practitioners. Since Dockerfiles are often not regarded as traditional source code, they are frequently excluded from automated analysis and mining efforts. As a result, developers tend to acquire only minimal, task-oriented knowledge rather than developing a deep understanding of secure configuration practices. As noted by Phillips et al., effective configuration work often goes unnoticed, as “if you are good, no one ever knows about it” [1].

Several supportive tools, such as the Docker extension for Visual Studio Code which has been adopted by millions of users aim to assist developers in writing Dockerfiles. However, the presence of these tools does not guarantee adherence to recommended best practices. Studies have shown that Dockerfiles available in public repositories, such as GitHub, contain substantially more rule violations compared to those written by experienced practitioners. These violations frequently result in misconfigurations that can introduce security vulnerabilities, performance inefficiencies, and operational risks.

To address these challenges, this study focuses on the automated detection of Dockerfile misconfigurations using Natural Language Processing (NLP) techniques. Dockerfiles exhibit characteristics similar to structured textual data, making them suitable for NLP-based analysis. This research conducts a systematic evaluation of NLP-based models for identifying misconfigurations, comparing their performance using key metrics such as accuracy, precision, recall, F1-score, and computational efficiency. Particular attention is given to the models' ability to generalize and detect previously unseen or novel misconfigurations, which is essential for real-world applicability.

By comparing transformer-based models such as RoBERTa and CodeBERT, this thesis aims to identify their respective strengths and limitations in the context of Dockerfile analysis. The findings are expected to contribute to the advancement of intelligent, automated tools for misconfiguration detection, ultimately helping to reduce configuration-related vulnerabilities and improve the security and reliability of containerized software systems within modern DevOps environments.

1.2 Problem Statement

Docker is widely used in modern software development, yet many developers do not consistently follow recommended configuration practices. As a result, Dockerfiles often contain misconfigurations that can introduce security vulnerabilities, reduce performance, and increase the risk of system compromise.

Most existing tools for Dockerfile analysis, such as Hadolint, Trivy, Dockle, and Checkov, rely on predefined rule-based static analysis. While these tools are effective in identifying known issues, they are limited by fixed rules and cannot adapt to new or complex misconfiguration patterns. In addition, the use of machine learning or NLP techniques for Dockerfile misconfiguration detection remains limited.

Recent advances in artificial intelligence have highlighted the potential of data-driven approaches for configuration analysis. Unlike rule-based tools, AI-based methods can learn from existing Dockerfiles, generalize to unseen misconfigurations, and reduce false detection rates. However, there is currently no widely adopted AI-based solution that reliably detects a broad range of Dockerfile misconfigurations in real-world scenarios. This limitation creates a clear need for robust automated detection methods using advanced NLP models, such as RoBERTa and CodeBERT, that can accurately identify diverse and previously unseen Dockerfile misconfigurations, thereby improving the security and reliability of containerized systems.

1.3 Objectives

The main objectives of thesis includes;

- To design and develop automated detection framework using either roBERTa or CodeBERT, and find out which one has better performance.
- 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.

1.4 Scope and Limitation

1.4.1 Scope

This study focuses on the comparison of transformer-based models, specifically RoBERTa and CodeBERT, for automated misconfiguration detection in Dockerfiles. The analysis is limited to static examination of Dockerfile content using NLP techniques. Model performance is evaluated using standard metrics, including accuracy, precision, recall, and F1-score, to determine the effectiveness of each approach in detecting misconfigurations.

1.4.2 Limitation

This research is restricted to Dockerfiles and does not consider runtime behavior or container-level security analysis. The dataset consists of publicly available Dockerfiles, which may not fully represent all real-world configurations. Only two transformer-based models are evaluated, while other NLP and deep learning methods are not included. Additionally, computational resource limitations may constrain the scale of experimentation and model tuning.

1.5 Report Organization

The Report organization of this thesis is as follows:

Unit 1: Introduction

The background and purpose of the study are explained in the introduction section. It explains the issue being tackled, the study's goals, the extent of the work, and the constraints faced during the investigation.

Unit 2: Background Study and Literature Review

The theoretical ideas, important research, and current techniques pertinent to the subject are described in the background study and literature review section. It explains the gaps in the literature and explains why the suggested method is necessary.

Unit 3: Methodology

The general research strategy is explained in the methodology section. The dataset, data pretreatment procedures, models or algorithms employed, and performance evaluation measures used to evaluate the model are all described.

Unit 4: Implementation

The practical application of the research is explained in the implementation section. It outlines the methods, instruments, and procedures used to prepare the data, train the model, and carry out testing and validation.

Unit 5: Result and Analysis

The findings of the experiments are described in the results and analysis section. It explains crucial findings, quantitative results, visual outputs, and how the model's performance was interpreted.

Unit 6: Conclusion and Future Recommendations

The study's general conclusions are presented in the conclusion and future recommendations section. It outlines the key contributions, draws attention to the drawbacks, and offers recommendations for future enhancements and additional study.

UNIT 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.
- **Incorrect or risky base image usage:** Using latest tag instead of pinned version, leading to non-reproducible builds.

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 (ICSE 2021) applied BERT-style models to detect broken Dockerfiles, while FLAKIDOCK (arXiv 2024) used LLMs to repair flaky builds. The Binnacle Project (Henkel et al., 2020) produced a Dockerfile corpus mainly for evaluating linters. 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.

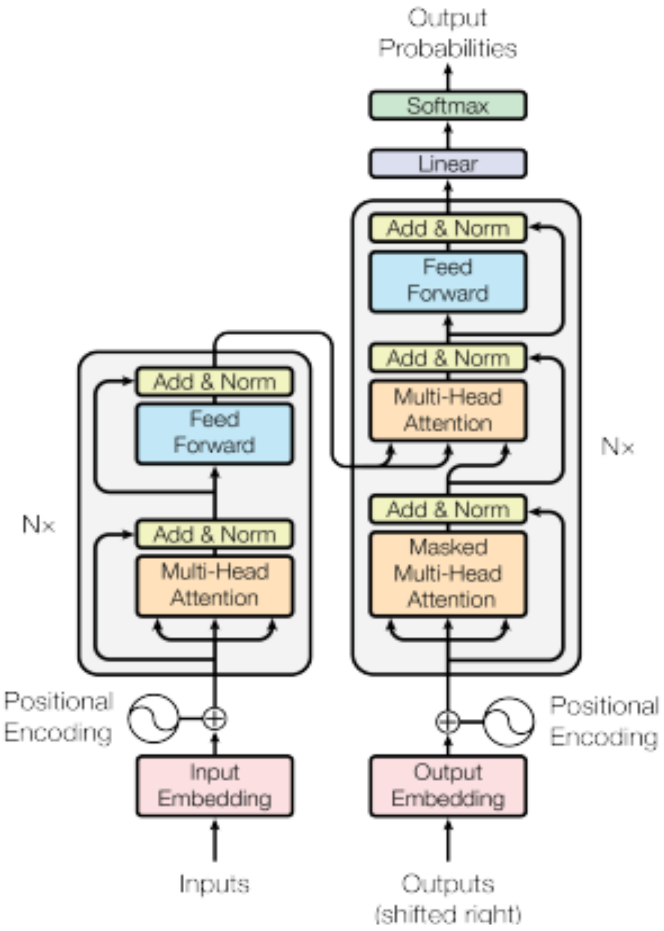


Figure 2.1:Transformer architecture [2]

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[3].

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[4].

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[5].

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[6].

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 [7].

The paper "DRIVE: Dockerfile Rule Mining and Violation Detection" proposes an approach to automatically mine implicit rules from Dockerfiles and detect potential violations [8]. 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.

UNIT 3: METHODOLOGY

3.1 Description of Methodology

The following diagram shows the overall system flowchart.

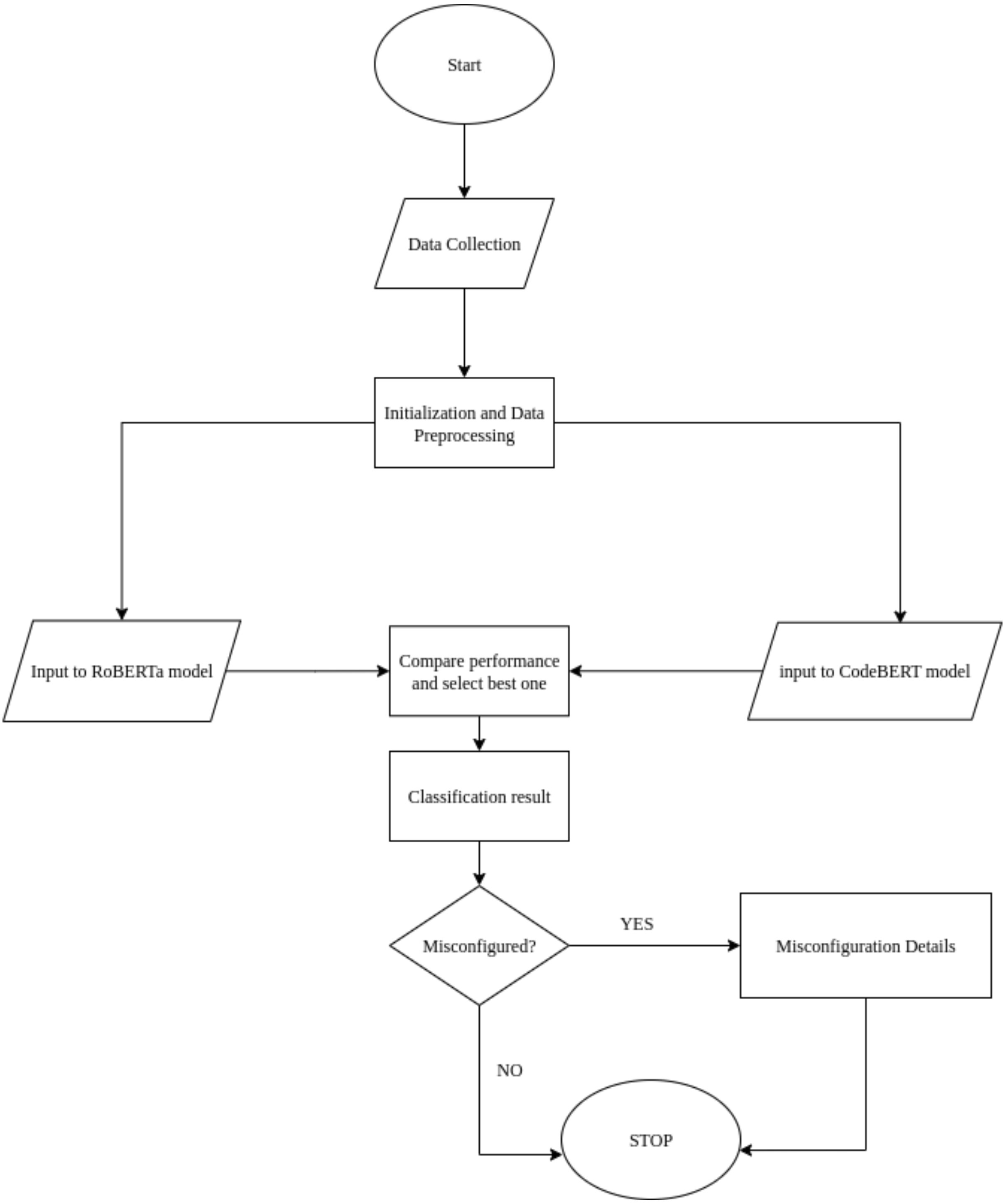


Figure 3.1: System flow chart

This flowchart represents an NLP-based automated misconfiguration detection system. The process starts with data collection, where Dockerfiles are gathered from relevant sources. Next, data initialization and preprocessing are performed, including steps such as comment removal, normalization, tokenization, and sequence formatting to prepare the input for model consumption. The preprocessed data is then passed as input embeddings to the NLP model for analysis.

The model performs contextual feature extraction and classification to determine whether a misconfiguration is present. Based on the classification output, the system evaluates the prediction. If a misconfiguration is detected, the system generates detailed misconfiguration reports, including the type and location of the issue. If no misconfiguration is identified, the process terminates without further action. In both cases, the system concludes after completing the inference process.

3.2 Dataset Description

The dataset used in this research was compiled from publicly available Dockerfiles. The primary source is the Dockerfile corpus proposed by Henkel et al., which consists of approximately 178,000 unique Dockerfiles extracted from GitHub repositories [9]. Introduced at ICSE 2020, this corpus reflects real-world Docker usage practices up to around 2019 and has been widely used in empirical studies related to software configuration quality, making it a reliable foundation for Dockerfile analysis.

However, Dockerfile syntax and recommended best practices have evolved considerably since 2020, particularly with the increased adoption of multi-stage builds, updated artifact management techniques, and enhanced security guidelines. To better capture current practices, additional Dockerfiles were collected using the GitHub Search API, targeting repositories actively updated between 2021 and 2025. This approach ensures that the dataset represents both historical and recent Dockerfile authoring trends.

All collected Dockerfiles were processed using line-level parsing, where individual instructions such as FROM, RUN, COPY, and WORKDIR were extracted for analysis. Each instruction was labeled as either correctly configured or misconfigured, forming the basis for supervised learning. Misconfigured instructions were further annotated at the rule level to enable fine-grained analysis.

In total, 178,857 Dockerfiles were included in the final dataset. After instruction-level extraction, the dataset comprised approximately 1.6 million Dockerfile commands. The data was divided into training (80%), validation (10%), and testing (10%) sets using stratified sampling to maintain consistent label distributions across all subsets. The detailed data splitting strategy is described in the Data Splitting section.

3.3 Data Preprocessing

A large-scale Dockerfile dataset was prepared by combining Henkel's publicly available Dockerfile corpus with additional Dockerfiles collected using the GitHub Search API. In total, 178,857 Dockerfiles were gathered from both sources. The dataset was designed to support a two-stage learning framework consisting of **binary misconfiguration detection (Stage 1)** and **rule-specific misconfiguration prediction (Stage 2)**.

3.3.1 Data preparation

The dataset preparation process involved the following steps:

Step 1: Data collection: Dockerfiles from Henkel's corpus were extracted and stored for analysis. To enhance dataset diversity and reflect recent practices, additional Dockerfiles were retrieved using the GitHub Search API. Each Dockerfile was tagged with its source (Henkel or GitHub) to ensure traceability.

Step 2: Line-level parsing:Each Dockerfile was parsed line by line, and individual Dockerfile commands were extracted. Each command was linked to its corresponding file identifier and line number, enabling detailed line-level analysis.

Step 3: Annotation and labeling: Each extracted command was annotated using a combination of rule-based checks and manual verification to determine whether it was correctly configured or misconfigured.

- For **Stage 1**, a binary label (label) was assigned to indicate whether a command was correct or misconfigured.
- For **Stage 2**, misconfigured commands were further assigned a multi-class label (label_rule) identifying the specific violated rule, along with a descriptive explanation (label_message).

Step 4: Feature extraction and dataset construction: Relevant features and metadata were extracted and stored in a structured CSV format. Each dataset record includes the following fields:

- source: Origin of the Dockerfile (Henkel or GitHub)
- file_id: Unique identifier for each Dockerfile
- line_number: Line number of the command within the Dockerfile
- line_content: Content of the Dockerfile command
- label: Binary indicator for correct or misconfigured commands (Stage 1 target)
- label_rule: Specific misconfiguration rule (Stage 2 target)
- label_message: Description of the violated rule

This unified dataset forms the foundation for both learning stages. While Stage 1 uses all labeled commands for binary classification, Stage 2 operates only on the subset of misconfigured commands with valid rule annotations.

3.3.2 Data splitting

To support efficient model training and fair evaluation, the dataset was divided into training, validation, and test sets using stratified sampling. Since this research follows a two-stage learning framework, separate data-splitting strategies were applied for each stage.

Stage 1: Binary misconfiguration classification

Stage 1 focuses on binary classification of Dockerfile commands as either correct or misconfigured. The original dataset contained approximately 1.6 million commands, with a nearly balanced class distribution of about 801,000 samples per class. Due to computational constraints, training on the full dataset was not feasible. Therefore, a controlled downsampling strategy was applied.

An equal number of samples were randomly selected from each class, with 200,000 commands per class, resulting in a reduced dataset of approximately 400,000 samples. This approach preserved class balance while reducing computational overhead.

The reduced dataset was then split into:

- Training set: 80%
- Validation set: 10%
- Test set: 10%

Stratified sampling based on the binary class label ensured consistent class distributions across all subsets. The split was performed in two steps by first separating the test set and then dividing the remaining data into training and validation sets. Each subset was stored as a separate CSV file and used consistently across all Stage 1 experiments.

Stage 2: Rule-specific misconfiguration classification

Stage 2 addresses rule-specific misconfiguration prediction and is formulated as a multi-class classification problem. This stage operates only on misconfigured commands and uses the rule label (label_rule) as the target variable.

Before splitting, the dataset was filtered to retain only misconfigured samples with valid rule annotations. Since rule distributions were highly imbalanced, a rule-aware balancing strategy was applied. Rules with fewer than 200 samples were excluded, while dominant rules were capped at 5,000 samples through random downsampling. This reduced extreme imbalance while maintaining meaningful rule diversity.

The balanced dataset was then shuffled and divided into:

- Training set: 80%
- Validation set: 10%
- Test set: 10%

Stratified sampling based on rule labels ensured proportional representation of each class across all subsets. As in Stage 1, the test set was separated first, followed by the training–validation split. The final datasets were saved as stage2_train.csv, stage2_val.csv, and stage2_test.csv and used consistently for all Stage 2 experiments.

3.3.3 Tokenization and encoding

To support transformer-based modeling, Dockerfile commands were converted into numerical representations using model-specific tokenization and encoding techniques. A unified tokenization pipeline was applied across both learning stages—binary misconfiguration detection (Stage 1) and rule-specific prediction (Stage 2)—to ensure consistent input representations, reproducibility, and fair comparison between models.

Both RoBERTa and CodeBERT operate on subword-level token sequences; therefore, their respective pretrained tokenizers were used. The microsoft/codebert-base tokenizer was employed for CodeBERT experiments, as it is optimized for source code and programming-related text. For RoBERTa experiments, the roberta-base tokenizer was used. In all cases, Dockerfile command lines were treated as code-like sequences and tokenized accordingly.

Tokenization was performed using a consistent configuration across all models and stages:

- Maximum sequence length: 256 tokens
- Truncation: Enabled for commands exceeding the maximum length
- Padding: Applied to a fixed length to ensure uniform input dimensions

This standardized configuration enabled efficient batch processing and ensured identical input dimensionality for both RoBERTa and CodeBERT, supporting a fair performance comparison.

For Stage 1, binary class labels were numerically encoded, where correctly configured commands were assigned the value 1 and misconfigured commands the value 0. For Stage 2, only misconfigured commands were included, and their corresponding rule labels were encoded as multi-class targets. To prevent information leakage, a label encoder was trained exclusively on the training set and subsequently applied to the validation and test sets. The fitted encoder was serialized and stored to ensure consistent label decoding during evaluation.

Given the large dataset size and hardware limitations, tokenization was conducted using a chunk-based, memory-efficient approach, where each dataset split was processed incrementally. This strategy prevented excessive memory usage and enabled large-scale preprocessing on commodity hardware.

The resulting tokenized outputs—including input IDs, attention masks, and encoded labels—were serialized and saved to disk in binary format. This eliminated the need for repeated tokenization during training, significantly reducing preprocessing overhead and improving training efficiency.

These pre-tokenized datasets were reused directly for model training and evaluation across both learning stages and for both RoBERTa and CodeBERT. By decoupling tokenization from the training loop, the experimental setup ensured consistent preprocessing, faster experimentation, and reproducible results.

3.3.4 Dataset summary

Table 3.1 Dataset summary

Aspect	**Stage 1**	**Stage 2**
Original dataset (after balanced)	16,04,322	Same dataset
Classes	2 (correct vs wrong)	63 rules
Class balance	2,00,000 per class	filtered <200 samples and capped at 5000 samples per rule
Total samples used	4,00,000	1,73,297
Training set (80%)	3,20,000	1,38,637
Validation set (10%)	40,000	17,330
Test set (10%)	40,000	17,330
Stratification criterion	Binary label	Rule label
Purpose	Detect if command misconfigured	Predict violated rule if misconfigured

3.4 Description of Algorithms / Mathematical Models

3.4.1 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,….,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.

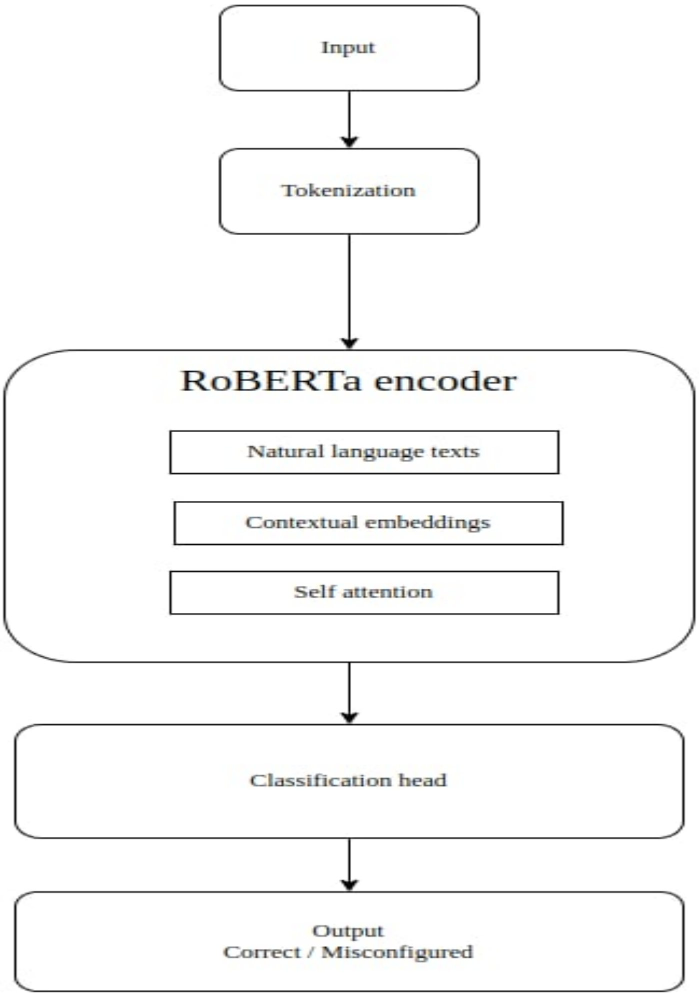


Figure 3.2:RoBERTa architecture

3.4.2 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 $X=\{x_1,x_2,...,x_n\}$ (where tokens may represent code or text), the encoder produces contextual embeddings $H=\{h_1,h_2,...,h_n\}$. 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_l is computed as:

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

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.

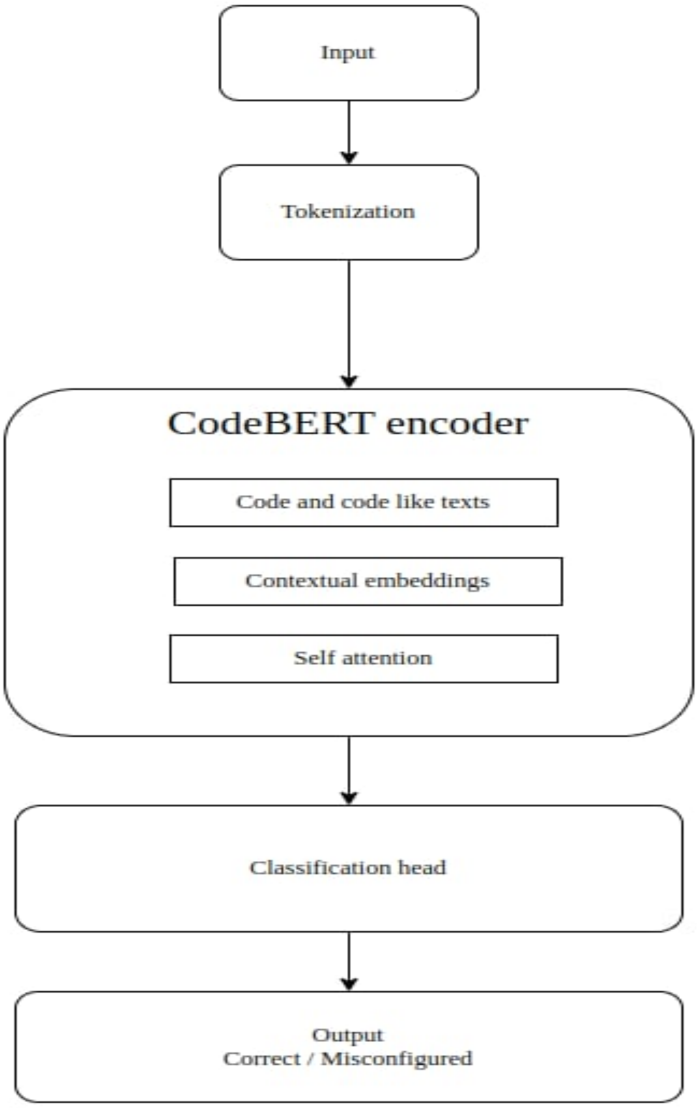


Figure 3.3: CodeBERT architecture

3.5 Performance / Evaluation Metrics:

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) were 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;

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

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;

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

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.

UNIT 4: IMPLEMENTATION

4.1 Tools and Techniques

4.1.1 Tools and technologies

To implement and evaluate the proposed models for misconfiguration detection, several python based libraries and tools were used. These tools facilitated data processing, model development, performance evaluation, and result visualization. Below is a list of key libraries used in this research:

- NumPy (Numerical Python)
- Matplotlib
- Pandas
- PyTorch
- Scikit learn
- Transformers

4.1.2 Environmental Setup

This research was conducted in the following experimental environment:

Hardware:

System: Personal laptop computer

RAM: 16 GB main memory, 16 GB swap memory

Disk Type: SSD, 512 GB

Processor: 11th Generation Intel(R) Core (TM) i5 @ 2.40 GHz

Software:

Operating System: Ubuntu 20.04 LTS, 64-bit

Programming Language: Python 3.8.10

Development Tools: VS Code, Cursor

4.2 Implementation Details

4.2.1 Building the model

Stage 1: Binary misconfiguration classification

In Stage 1, the objective was to classify Dockerfile commands as either correctly configured or misconfigured. Two pretrained transformer models, CodeBERT and RoBERTa, were employed for this task. Each model was adapted for binary classification by appending a fully connected output layer with two neurons followed by a softmax activation function.

Dockerfile commands were tokenized into subword sequences with a maximum length of 256 tokens. To support memory-efficient training, the dataset was preprocessed and divided into smaller chunks, with approximately 200,000 samples per class selected for training.

Model training was performed using the AdamW optimizer with a learning rate of 2×10^{-5} for both models. Batch sizes were set to 16 for CodeBERT and 32 for RoBERTa, with gradient accumulation applied when required to accommodate hardware constraints. Model checkpoints were saved at the end of each epoch, including the model parameters, optimizer state, learning rate scheduler, and tokenizer, enabling training to be resumed if interrupted.

Device-aware training was implemented to support execution on GPU, Apple Metal Performance Shaders (MPS), or CPU, depending on hardware availability. Both models were trained for three epochs, allowing sufficient convergence and effective feature learning for distinguishing between correct and misconfigured Dockerfile commands. The trained models from this stage served as the foundation for the rule-specific classification task in Stage 2.

Stage 2: Rule specific misconfiguration classification

Stage 2 focused on identifying the specific misconfiguration rule violated by a Dockerfile command. This task was formulated as a multi-class classification problem and applied only to commands labeled as misconfigured. CodeBERT and RoBERTa were adapted by modifying their output layers to match the number of unique misconfiguration rules present in the dataset.

Input commands were tokenized into subword sequences with a maximum length of 256 tokens. To ensure efficient training within hardware limitations, the dataset was again processed in memory-efficient chunks.

Training was conducted using the AdamW optimizer with a learning rate of 5×10^{-5}, a batch size of 32, and a linear learning rate scheduler. Model checkpoints were saved after each epoch, including all necessary training components to allow seamless resumption of training. Training was executed on GPU, Apple MPS, or CPU, with real-time progress tracking implemented using tqdm.

Both models were trained for three epochs on the rule-labeled dataset. By focusing exclusively on misconfigured commands, this stage enabled the models to learn fine-grained distinctions between different types of configuration violations. The training pipeline ensured efficient handling of the multi-class dataset and robust learning for accurate rule prediction.

UNIT 5: RESULT AND ANALYSIS

5.1 Findings and Observations

5.1.1 Training and validation loss

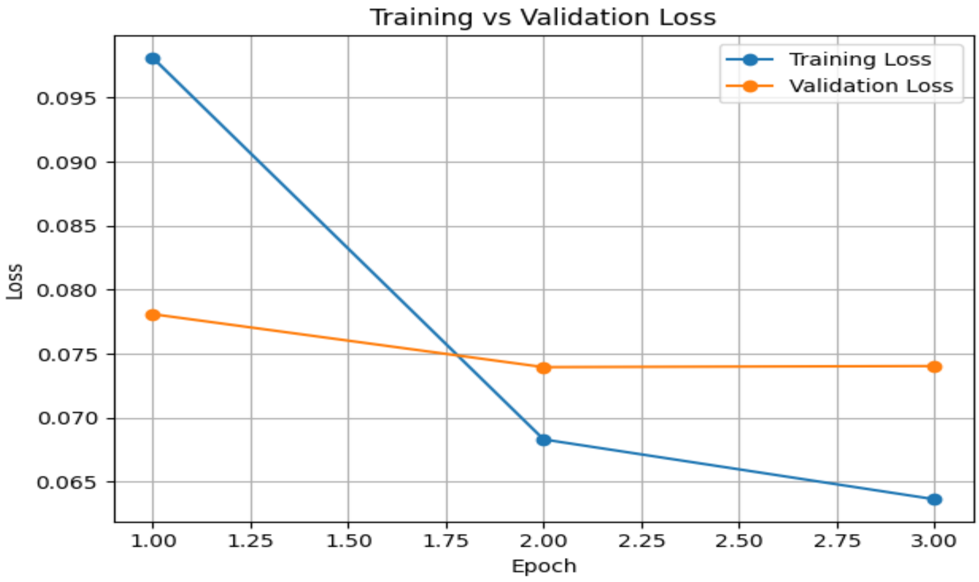


Figure 5.1: Training vs validation loss for CodeBERT

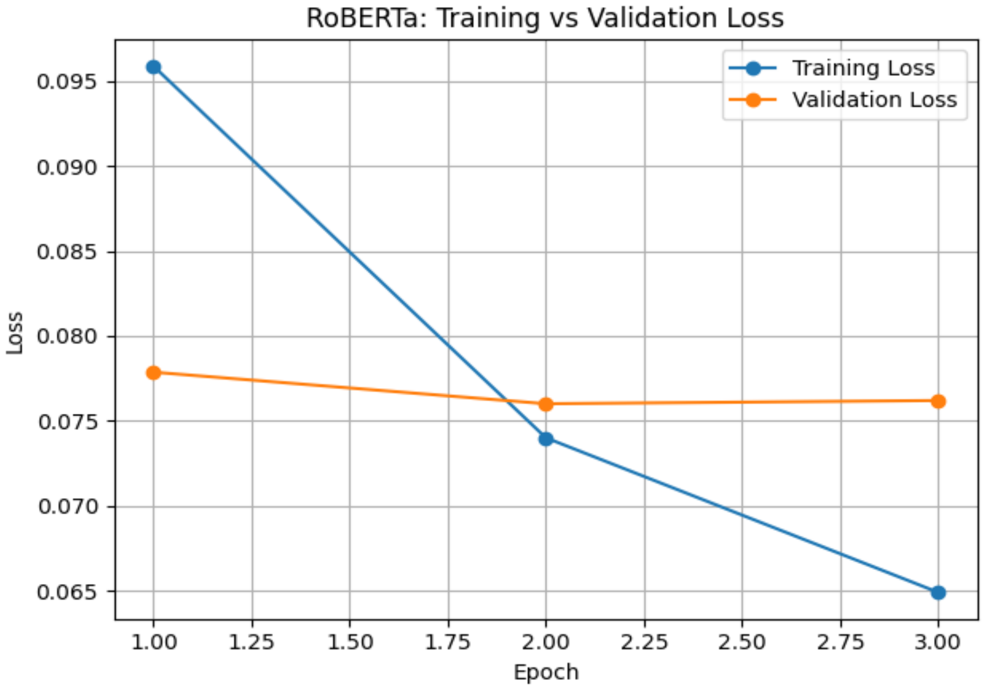


Figure 5.2: Training vs validation loss for RoBERTa

5.1.2 Validation performance

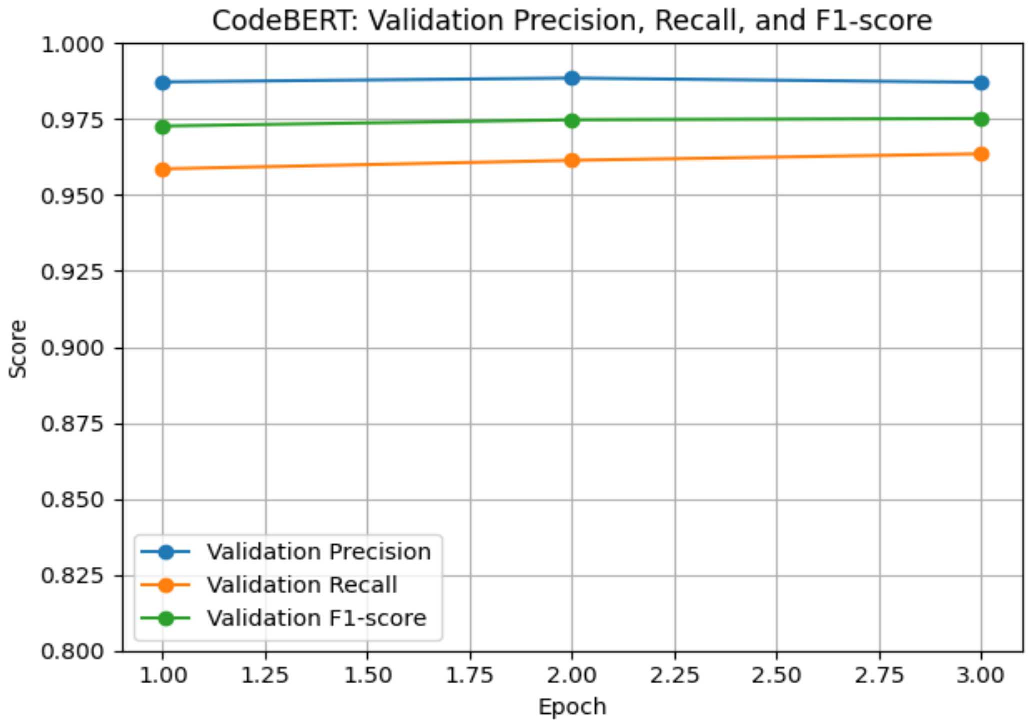


Figure 5.3: Validation performance for CodeBERT

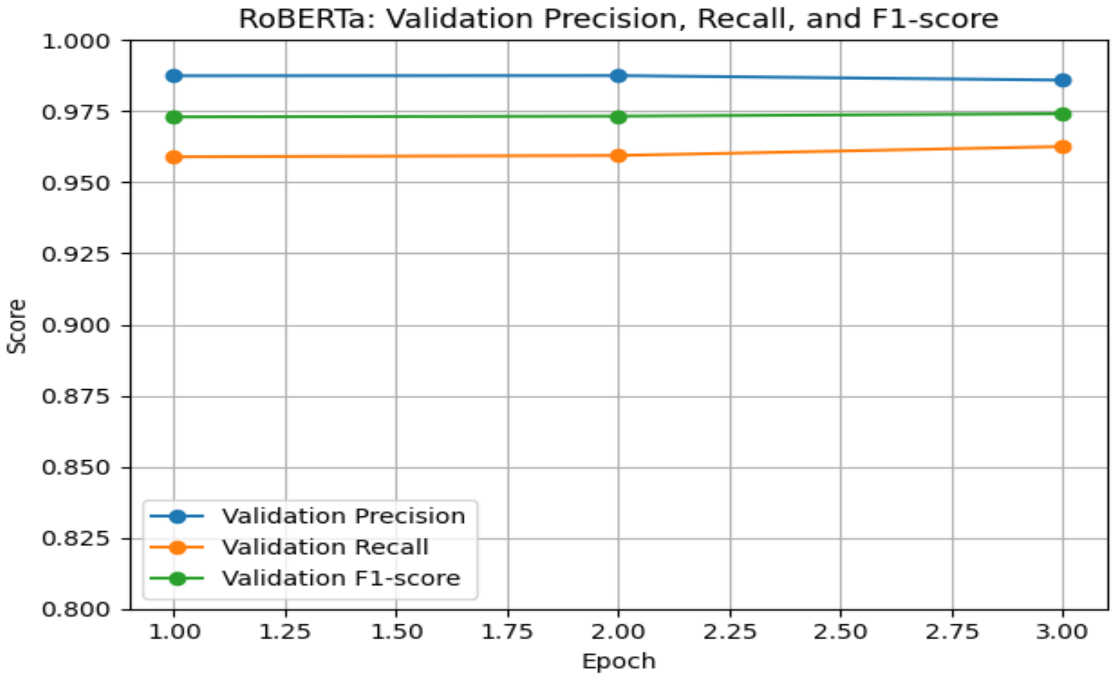


Figure 5.4: Validation performance for RoBERTa

5.1.3 Classification performance

Binary misconfiguration classification

Table 5.1: Performance metrics of CodeBERT for binary misconfiguration classification

Class	Precision	Recall	F1-Score
Correct	0.97	0.99	0.98
Wrong	0.99	0.97	0.98

Overall accuracy: 0.98, ROC AUC: 0.997

Table 5.2: Performance metrics of RoBERTa for binary misconfiguration classification

Class	Precision	Recall	F1-Score
Correct	0.98	0.99	0.99
Wrong	0.99	0.98	0.98

Overall accuracy: 0.98, ROC AUC: 0.998

Table 5.3: Confusion matrix of CodeBERTa for binary misconfiguration classification

Actual/Predicted	Correct	Wrong
Correct	19,758	242
Wrong	677	19,323

Table 5.4: Confusion matrix of RoBERTa for binary misconfiguration classification

Actual/Predicted	Correct	Wrong
Correct	19,706	294
Wrong	679	19,321

Rule specific misconfiguration classification

Table 5.5: Overall performance metrics of CodeBERT for rule specific misconfiguration classification

Metric	**Value**
Accuracy	0.7070
Macro-averaged F1	0.7136
Weighted-averaged F1	0.7040
Top-1 accuracy	0.7070
Top-3 accuracy	0.8938
Macro-averaged ROC-AUC (One-vs-Rest)	0.9823
Weighted-averaged ROC-AUC (One-vs-Rest)	0.9816

Table 5.6: Overall performance metrics of RoBERTa for rule specific misconfiguration classification

Metric	**Value**
Accuracy	0.7048
Macro-averaged F1	0.7055
Weighted-averaged F1	0.7013
Top-1 accuracy	0.7048
Top-3 accuracy	0.8901
Macro-averaged ROC-AUC (One-vs-Rest)	0.9824
Weighted-averaged ROC-AUC (One-vs-Rest)	0.9815

5.2 Result Analysis

The model performance in the experiment was evaluated using various experiments and metrics.

5.2.1 Accuracy

After completion of training the model, the accuracy of CodeBERT and RoBERTa for binary misconfiguration classification and rule specific misconfiguration classification is given in the table below.

Table 5.7 Accuracy comparison of the models

Model	Accuracy	
	Binary classification	**Multi class classification**
CodeBERT	0.98	0.7070
RoBERTa	0.98	0.7048

Table 5.7 compares the accuracy of CodeBERT and RoBERTa on binary and multi-class (rule-specific) misconfiguration classification tasks. Both models achieve 0.98 accuracy on the binary task, while for multi-class classification, CodeBERT outperforms RoBERTa (70.70% vs 70.48%). This indicates that both models are highly effective for binary detection, and that further improvements for rule-specific classification may require additional data or task-specific adaptations.

5.2.2 Classification performance

Table 5.8 Binary classification performance of the models

Model / Metrics	Precision	Recall	F1 score	AUC-ROC
CodeBERT	0.98	0.98	0.98	0.9970
RoBERTa	0.98	0.98	0.98	0.9968

Table 5.8 compares the overall performance of CodeBERT and RoBERTa for binary misconfiguration classification. Both models achieve identical precision, recall, and F1-score (0.98), indicating comparable ability to correctly classify both classes. CodeBERT shows a

higher ROC AUC (0.9970 vs 0.9968), suggesting marginally better discriminative performance, though overall the models perform very similarly.

Table 5.9 Overall performance comparison of models for rule specific misconfiguration classification of the models

Metric	CodeBERT	RoBERTa
Accuracy	0.7070 (70.70%)	0.7048 (70.48%)
Macro-averaged F1	0.7136 (71.36%)	0.7055 (70.55%)
Weighted-averaged F1	0.7040 (70.40%)	0.7013 (70.13%)
Top-1 Accuracy	0.7070 (70.70%)	0.7048 (70.48%)
Top-3 Accuracy	0.8938 (89.38%)	0.8901 (89.01%)
Macro ROC-AUC (One-vs-Rest)	0.9823 (98.23%)	0.9824 (98.24%)
Weighted ROC-AUC (One-vs-Rest)	0.9816 (98.16%)	0.9815 (98.15%)

Table 5.9 compares the overall performance of CodeBERT and RoBERTa for rule-specific (Stage-2) misconfiguration classification. CodeBERT outperforms RoBERTa in accuracy (70.70% vs 70.48%) and F1-scores (Macro F1: 71.36% vs 70.55%; Weighted F1: 70.40% vs 70.13%), indicating a marginally better ability to capture rule-specific misconfigurations. Top-3 accuracy and ROC-AUC values are nearly identical, suggesting both models have similar overall discriminative capability. Overall, the differences are small, and both transformer-based models demonstrate competitive performance for multi-class misconfiguration classification.

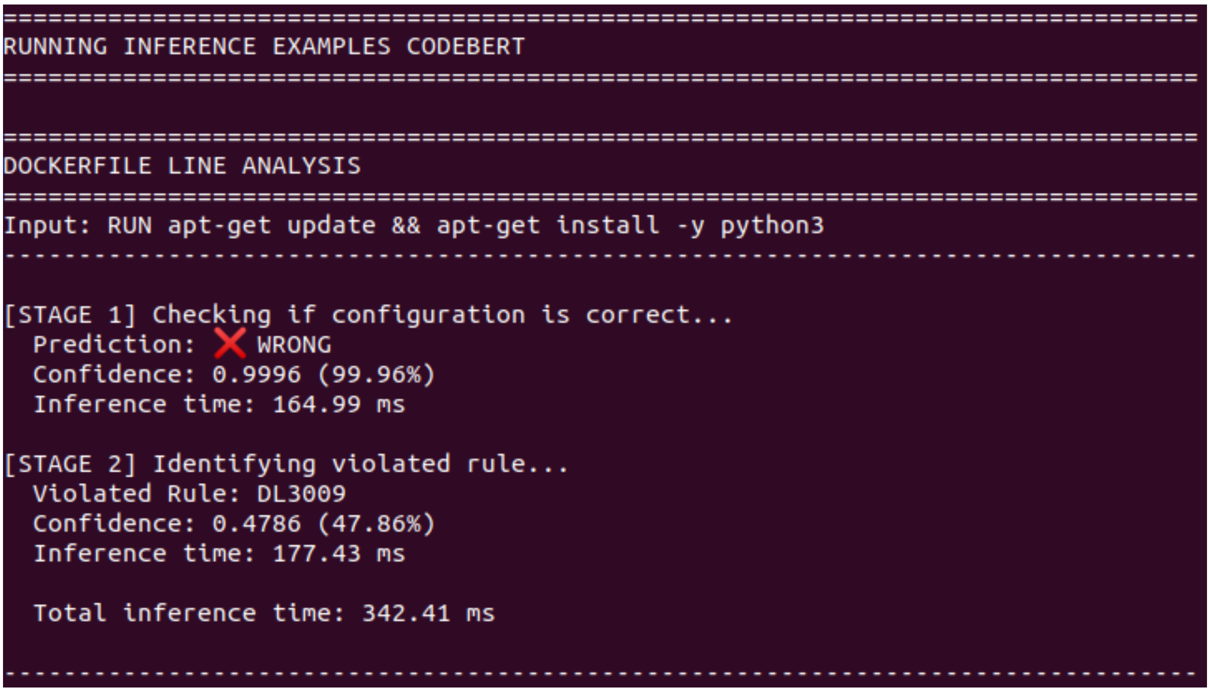
Table 5.10 Per class performance comparison of models for rule specific misconfiguration classification of the models

Metric	**CodeBERT**	**RoBERTa**
Total number of rules	63	63
Average F1 Score	0.7136	0.7055
Median F1 score	0.7143	0.7027
Best F1 Score (Rule)	1.0000 (DL3007)	1.0000 (DL3007)
Worst F1 Score (Rule)	0.2322 (DL3008)	0.1132 (DL3036)
Number of rules with F1 > 0.8	21	21
Number of rules with F1 < 0.5	11	11

Table 5.10 summarizes the per-class performance of CodeBERT and RoBERTa for Stage-2 rule-specific misconfiguration classification. CodeBERT achieves higher average and median F1-scores (0.7136 and 0.7143) compared to RoBERTa (0.7055 and 0.7027), indicating a marginally better overall ability to classify rules correctly. Both models share the same best-performing rule (DL3007), while their worst-performing rules differ, with RoBERTa showing a lower minimum F1-score (0.1132 vs 0.2322). The number of rules with high F1-scores (>0.8) and low F1-scores (<0.5) is identical for both models, suggesting similar distribution of performance across rules. Overall, CodeBERT demonstrates better per-class performance, but both models exhibit comparable strengths and weaknesses across the 63 rules.

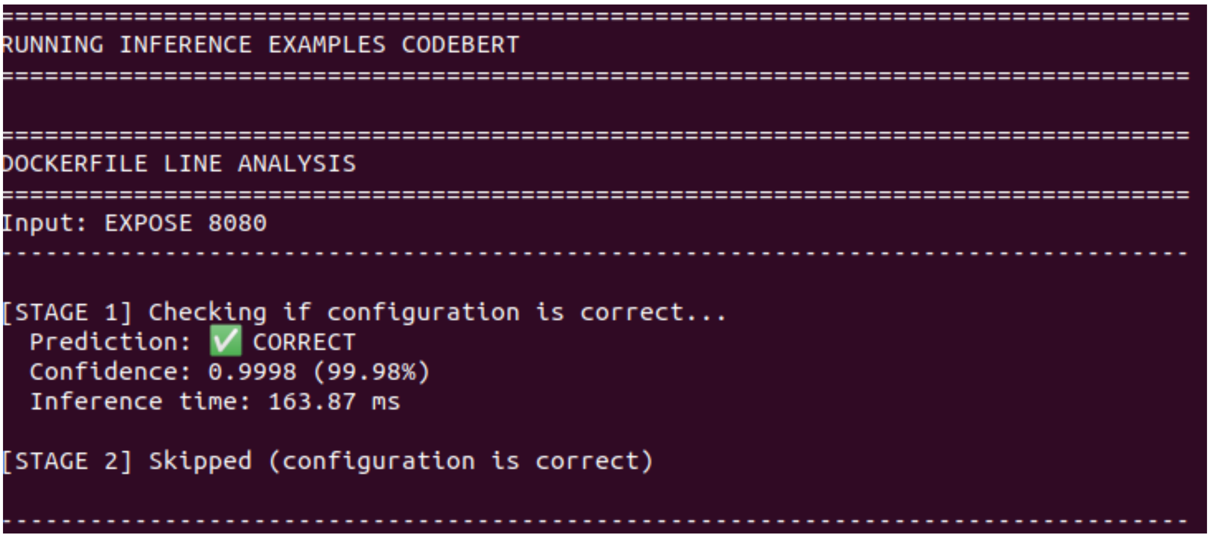
5.2.3 Inference

Both correct and incorrectly configured data were fed to the models for classification and prediction of the violated rules.



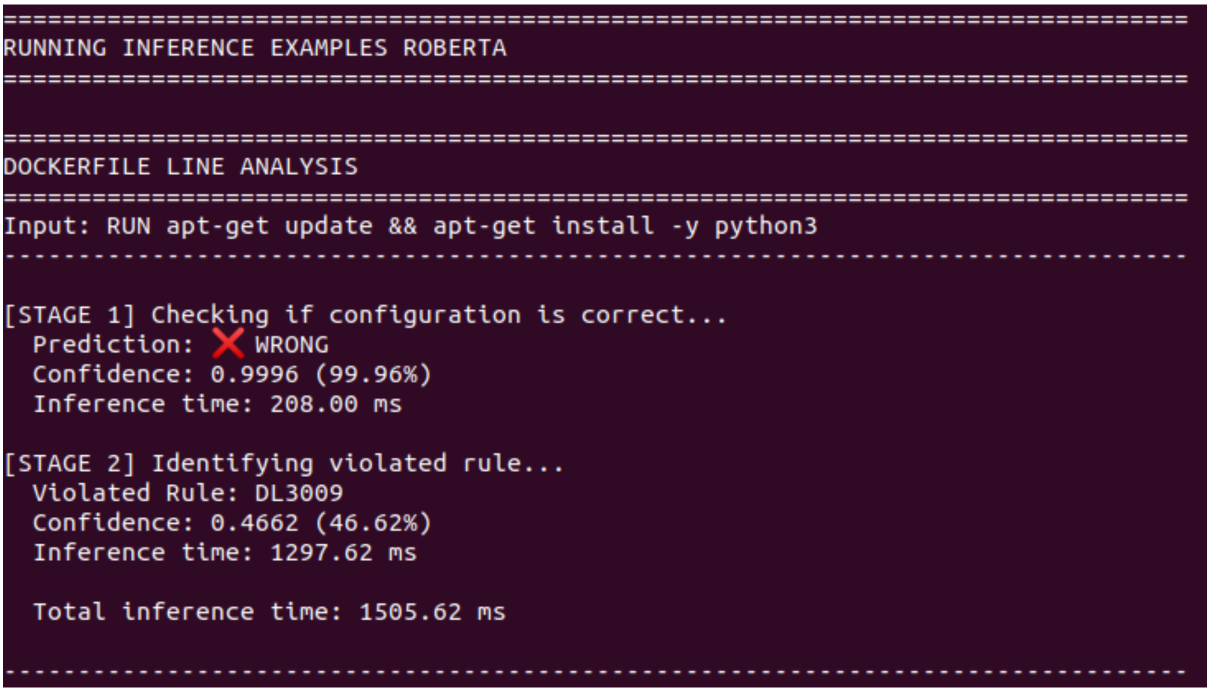
```
================================================================================
RUNNING INFERENCE EXAMPLES CODEBERT
================================================================================

================================================================================
DOCKERFILE LINE ANALYSIS
================================================================================
Input: RUN apt-get update && apt-get install -y python3
--------------------------------------------------------------------------------

[STAGE 1] Checking if configuration is correct...
  Prediction: ❌ WRONG
  Confidence: 0.9996 (99.96%)
  Inference time: 164.99 ms

[STAGE 2] Identifying violated rule...
  Violated Rule: DL3009
  Confidence: 0.4786 (47.86%)
  Inference time: 177.43 ms

  Total inference time: 342.41 ms

--------------------------------------------------------------------------------
```

Figure 5.5: CodeBERT inference on misconfigured command



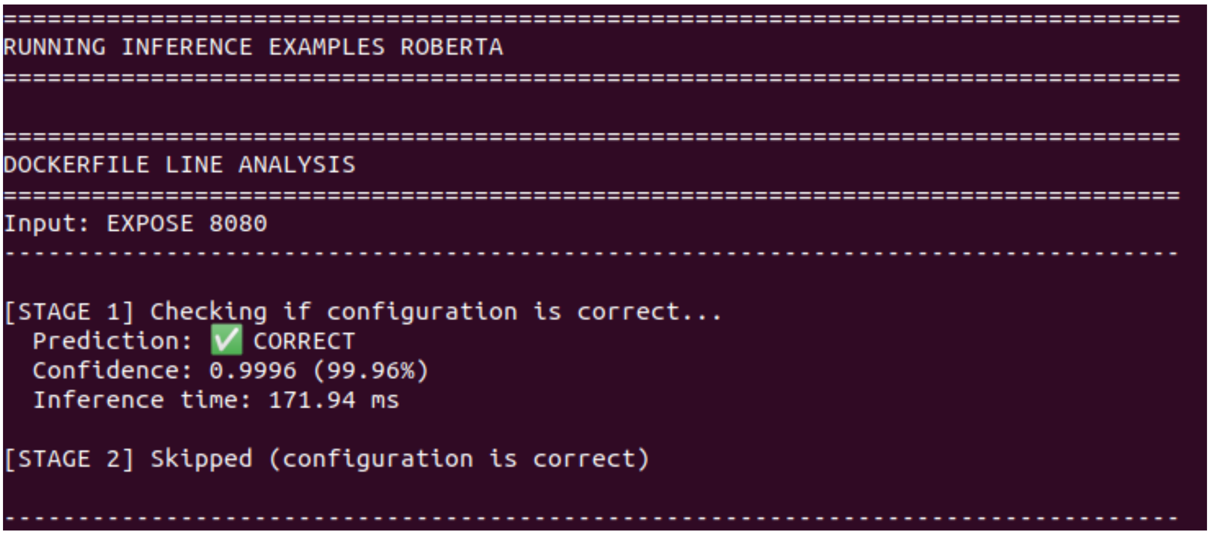
```
================================================================================
RUNNING INFERENCE EXAMPLES CODEBERT
================================================================================

================================================================================
DOCKERFILE LINE ANALYSIS
================================================================================
Input: EXPOSE 8080
--------------------------------------------------------------------------------

[STAGE 1] Checking if configuration is correct...
  Prediction: ✅ CORRECT
  Confidence: 0.9998 (99.98%)
  Inference time: 163.87 ms

[STAGE 2] Skipped (configuration is correct)

--------------------------------------------------------------------------------
```

Figure 5.6: CodeBERT inference on correctly configured command



```
================================================================================
RUNNING INFERENCE EXAMPLES ROBERTA
================================================================================

================================================================================
DOCKERFILE LINE ANALYSIS
================================================================================
Input: RUN apt-get update && apt-get install -y python3
--------------------------------------------------------------------------------

[STAGE 1] Checking if configuration is correct...
  Prediction: ❌ WRONG
  Confidence: 0.9996 (99.96%)
  Inference time: 208.00 ms

[STAGE 2] Identifying violated rule...
  Violated Rule: DL3009
  Confidence: 0.4662 (46.62%)
  Inference time: 1297.62 ms

  Total inference time: 1505.62 ms

--------------------------------------------------------------------------------
```

Figure 5.7: RoBERTa inference on misconfigured command



```
================================================================================
RUNNING INFERENCE EXAMPLES ROBERTA
================================================================================

================================================================================
DOCKERFILE LINE ANALYSIS
================================================================================
Input: EXPOSE 8080
--------------------------------------------------------------------------------

[STAGE 1] Checking if configuration is correct...
  Prediction: ✅ CORRECT
  Confidence: 0.9996 (99.96%)
  Inference time: 171.94 ms

[STAGE 2] Skipped (configuration is correct)

--------------------------------------------------------------------------------
```

Figure 5.8: RoBERTa inference on correctly configured command

UNIT 6: CONCLUSION AND FUTURE RECOMMENDATIONS

6.1 Conclusion

This thesis explored the use of transformer-based models, CodeBERT and RoBERTa, for automated detection and classification of misconfigurations in Dockerfiles. The study was structured into two stages: binary misconfiguration detection (Stage 1) and rule-specific misconfiguration classification (Stage 2).

In Stage 1, both CodeBERT and RoBERTa demonstrated strong performance in distinguishing correctly configured Dockerfile commands from misconfigured ones. The two models achieved identical overall accuracy of 0.98, with macro-averaged and weighted precision, recall, and F1-scores also reaching 0.98. In addition, both models produced very high ROC–AUC values (0.9970 for CodeBERT and 0.9968 for RoBERTa), indicating excellent discriminative capability. These results confirm that transformer-based models are highly effective for binary misconfiguration detection.

In Stage 2, which focused on predicting the specific misconfiguration rule violated by a Dockerfile command, performance decreased as expected due to the increased complexity of multi-class classification. CodeBERT outperformed RoBERTa, achieving higher overall accuracy (70.70% vs. 70.48%) and a higher average F1-score (0.7136 vs. 0.7055). Both models exhibited comparable Top-3 accuracy and ROC–AUC values, suggesting similar effectiveness in capturing rule-level violations. Per-class analysis showed that frequently occurring rules were detected reliably by both models, while performance varied for less common or more complex rules.

Overall, the findings indicate that both CodeBERT and RoBERTa are highly effective for binary misconfiguration detection, with CodeBERT demonstrating marginally better performance in rule-specific classification. These results highlight the potential of transformer-based models as scalable and automated solutions for detecting misconfigurations in Infrastructure-as-Code artifacts. Such approaches can help reduce human error, improve configuration quality, and enhance the reliability of containerized systems. Future work may focus on improving performance for rare or complex rules, incorporating data augmentation techniques, or exploring ensemble and hybrid models to further improve classification accuracy.

6.2 Future Recommendation

Future work may focus on improving rule-specific classification by leveraging larger and more diverse datasets, as well as applying data augmentation or synthetic data generation techniques to enhance model performance on rare and complex misconfiguration rules.

Combining multiple transformer-based models through ensemble methods, or integrating learned models with rule-based heuristics, could further improve detection accuracy and robustness, particularly for rules that currently exhibit lower F1-scores.

In addition, deploying the proposed models within CI/CD pipelines would enable real-time misconfiguration detection, allowing potential issues to be identified and resolved before deployment. Such integration could significantly reduce configuration-related errors in production environments.

Integrate the detection framework as a pre-commit Git hook to provide instant feedback on misconfigurations and best practice violations.

Finally, the proposed approach can be extended to other Infrastructure-as-Code (IaC) technologies, such as Terraform, AWS CloudFormation, and Ansible. Expanding the methodology to these platforms would increase the generalizability and practical impact of automated, AI-driven misconfiguration detection across diverse DevOps ecosystems.

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

[3] 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

[4] 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

[5] 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

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

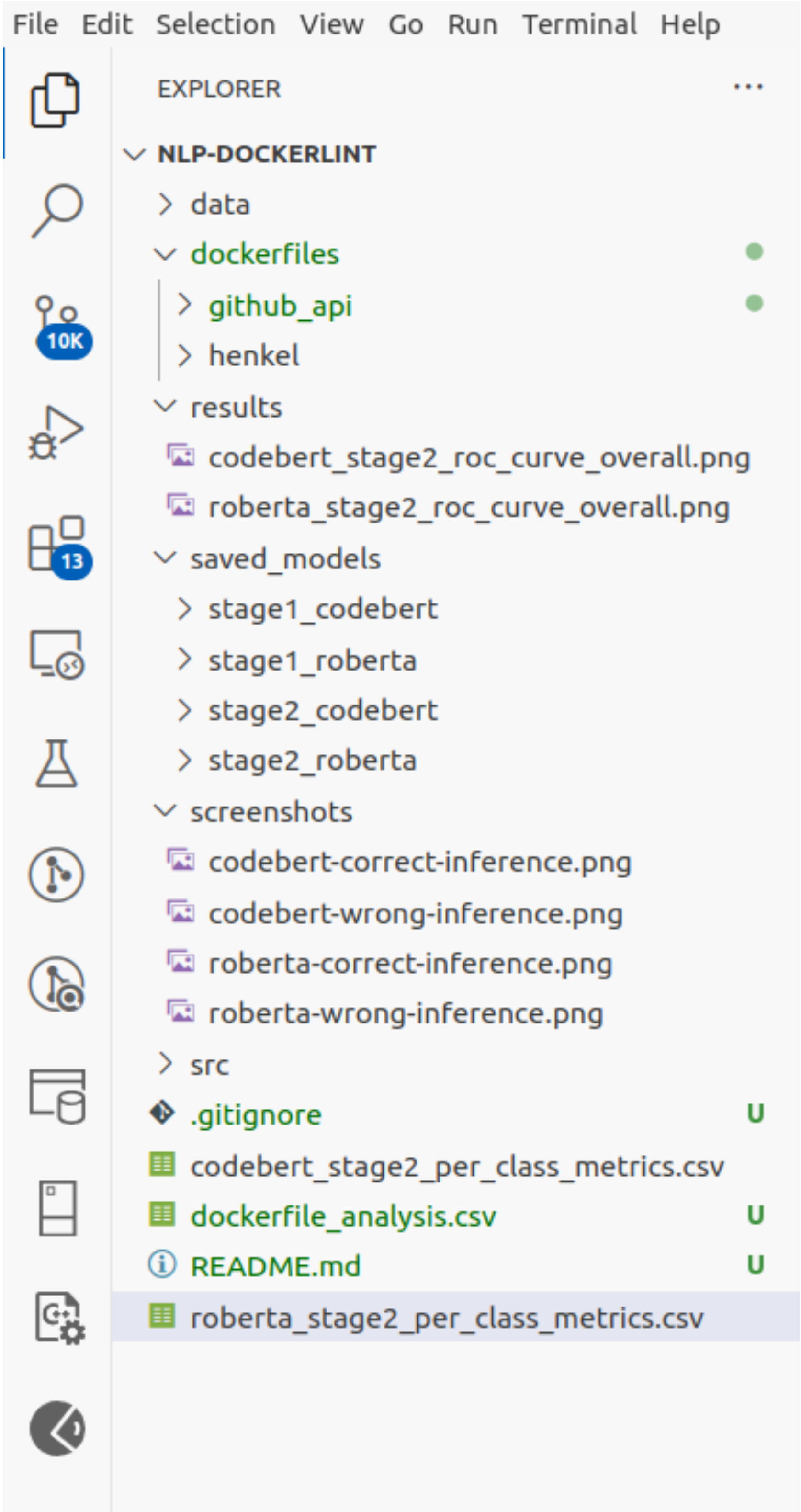
[7] G. Rosa, S. Scalabrino, G. Robles, and R. Oliveto, *Not all Dockerfile Smells are the Same: An Empirical Evaluation of Hadolint Writing Practices by Experts*. 2024. doi: 10.1145/3643991.3644905.

[8] 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

[9] 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.

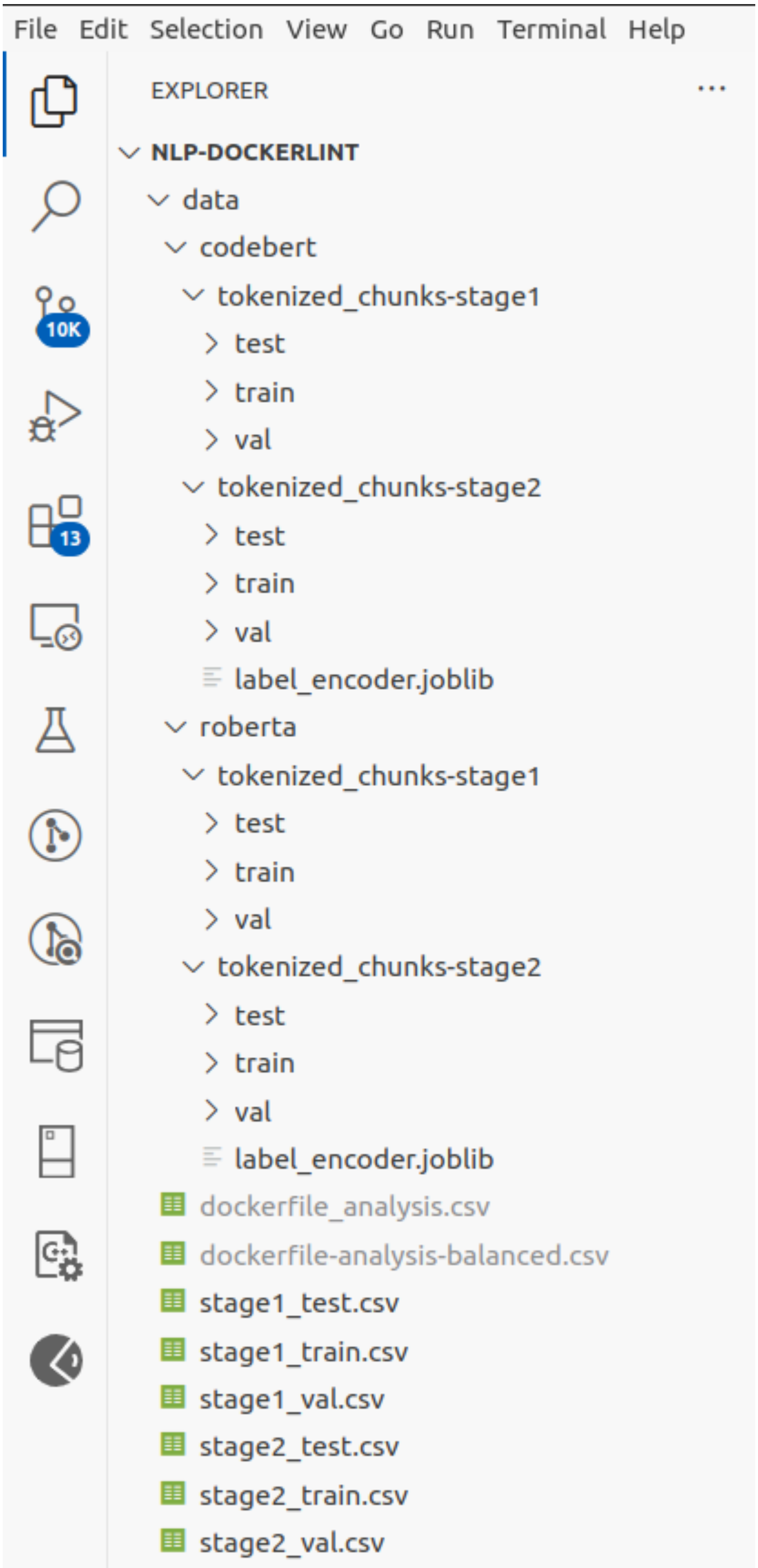
APPENDICES

Project directory structure (top level folders)



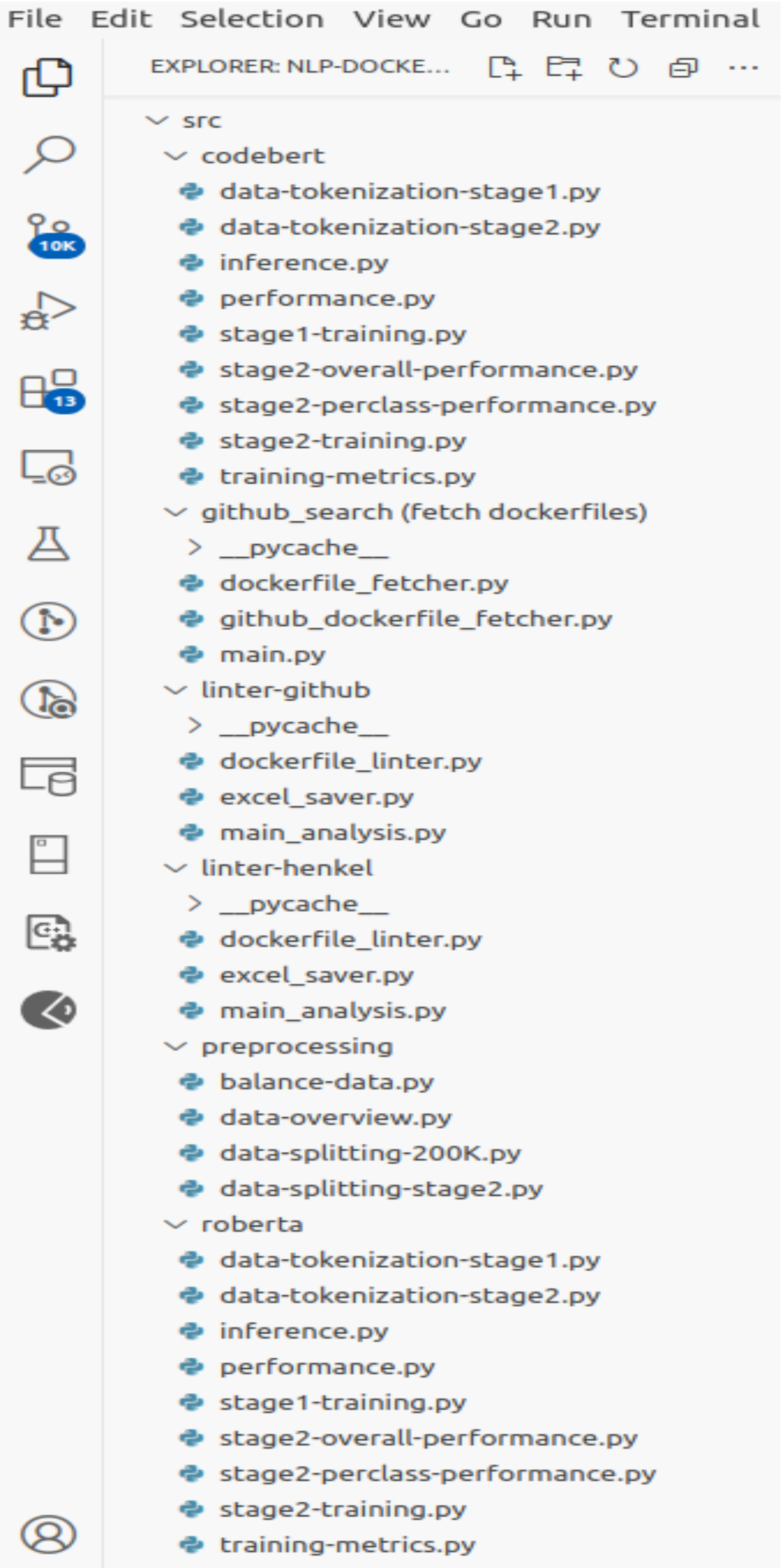
```
NLP-DOCKERLINT
  data
  dockerfiles                                   ●
    github_api                                  ●
    henkel
  results
    codebert_stage2_roc_curve_overall.png
    roberta_stage2_roc_curve_overall.png
  saved_models
    stage1_codebert
    stage1_roberta
    stage2_codebert
    stage2_roberta
  screenshots
    codebert-correct-inference.png
    codebert-wrong-inference.png
    roberta-correct-inference.png
    roberta-wrong-inference.png
  src
  .gitignore                                    U
  codebert_stage2_per_class_metrics.csv
  dockerfile_analysis.csv                       U
  README.md                                     U
  roberta_stage2_per_class_metrics.csv
```

Data directory structure



```
NLP-DOCKERLINT
  data
    codebert
      tokenized_chunks-stage1
        test
        train
        val
      tokenized_chunks-stage2
        test
        train
        val
        label_encoder.joblib
    roberta
      tokenized_chunks-stage1
        test
        train
        val
      tokenized_chunks-stage2
        test
        train
        val
        label_encoder.joblib
    dockerfile_analysis.csv
    dockerfile-analysis-balanced.csv
    stage1_test.csv
    stage1_train.csv
    stage1_val.csv
    stage2_test.csv
    stage2_train.csv
    stage2_val.csv
```

Source code directory structure



```
src
  codebert
    data-tokenization-stage1.py
    data-tokenization-stage2.py
    inference.py
    performance.py
    stage1-training.py
    stage2-overall-performance.py
    stage2-perclass-performance.py
    stage2-training.py
    training-metrics.py
  github_search (fetch dockerfiles)
    > __pycache__
    dockerfile_fetcher.py
    github_dockerfile_fetcher.py
    main.py
  linter-github
    > __pycache__
    dockerfile_linter.py
    excel_saver.py
    main_analysis.py
  linter-henkel
    > __pycache__
    dockerfile_linter.py
    excel_saver.py
    main_analysis.py
  preprocessing
    balance-data.py
    data-overview.py
    data-splitting-200K.py
    data-splitting-stage2.py
  roberta
    data-tokenization-stage1.py
    data-tokenization-stage2.py
    inference.py
    performance.py
    stage1-training.py
    stage2-overall-performance.py
    stage2-perclass-performance.py
    stage2-training.py
    training-metrics.py
```

Top 10 best performing rules (by F1 score) - CodeBERT

S.no	Rule name	Precision	Recall	F1 score
1	DL3007	1.000	1.000	1.000
2	DL3006	0.996	1.000	0.9980
3	DL3000	0.995	1.000	0.9976
4	DL4000	0.992	1.000	0.9960
5	DL3002	0.991	1.000	0.9954
6	DL3048	0.990	1.000	0.9951
7	DL3020	0.988	1.000	0.9940
8	DL3025	0.988	0.990	0.9890
9	DL3045	0.985	0.981	0.9834
10	DL3010	1.000	0.966	0.9830

Top 10 best performing rules (by f1 score) - RoBERTa

S.no	**Rule name**	**Precision**	**Recall**	**F1 score**
1	DL3007	1.000	1.000	1.000
2	DL3006	0.998	1.000	0.9990
3	DL3000	0.995	1.000	0.9976
4	DL4000	0.992	1.000	0.9960
5	DL3048	0.990	1.000	0.9951
6	DL3002	0.986	1.000	0.9932
7	DL3020	0.986	1.000	0.9930
8	DL3025	0.986	0.992	0.9890
9	DL3045	0.987	0.979	0.9834
10	DL3010	1.000	0.966	0.9830

Comparison of RoBERTa and CodeBERTa

This page explains the architectural and training differences between our two primary models.

RoBERTa (Robustly Optimized BERT Approach)

RoBERTa is a transformer-based model designed for high-performance natural language understanding. It optimizes the original BERT architecture by using larger datasets and improved training strategies to better capture contextual information.

- Training Data: Pre-trained on large English corpora, including BooksCorpus and English Wikipedia (~160GB).
- Parameters: Available in Base (125M) and Large (355M) configurations.
- Objective: Masked Language Modeling (MLM).
- Use Case: General NLP tasks such as text classification, Question Answering (QA), and Named Entity Recognition (NER).

CodeBERT

CodeBERT is a bimodal transformer-based model specifically designed to handle both natural language and programming language understanding. It extends the BERT architecture to capture the semantic relationships between code and its documentation.

- Training Data: Pre-trained on 6 programming languages and their corresponding documentation (~2.1M code–doc pairs).
- Parameters: Base (125M).
- Objective: Masked Language Modeling (MLM) and Replaced Token Detection (RTD).
- Use Case: Code search, summarization, and defect or misconfiguration detection in code-like structures.

Reason for Selection of RoBERTa and CodeBERT

Dockerfiles are unique artifacts that are neither purely natural language nor standard source code. RoBERTa and CodeBERT were selected to evaluate whether a general text understanding approach or a code-specific understanding approach is more effective for automated misconfiguration detection.

Performance Metrics

To provide a comprehensive evaluation of the Stage 2 multi-class classification, the following metrics were utilized:

- Macro-averaged F1-score: This calculates the F1-score for each of the 63 rules independently and then takes the unweighted mean . it treats all rules equally, regardless of how many samples each rule has in the dataset.
- Weighted-averaged F1-score: Similar to the macro-average, but it weights the F1-score of each rule by its presence (support) in the dataset. This provides a metric that accounts for class imbalance.
- Top-1 Accuracy: The standard accuracy metric where a prediction is counted as correct only if the model's highest-probability class matches the ground truth label.
- Top-3 Accuracy: A more lenient metric where a prediction is considered correct if the true rule label is among the top three highest probabilities predicted by the model. This is useful for evaluating model performance on rules with high semantic similarity.
- AUC-ROC (Area Under the Receiver Operating Characteristic Curve):
    - The ROC curve plots the True Positive Rate (Recall) against the False Positive Rate at various threshold values.
    - The AUC score measures the model's capacity to differentiate between classes, where a score of 1 represents perfect classification and 0.5 indicates random guessing.
    - A higher AUC score indicates better performance in distinguishing between correctly configured and misconfigured commands.

Dockerlint Rules

In Stage 2 of this research, the models were trained to identify 63 specific misconfiguration rules. Below are some of the fundamental rules encountered in the dataset:

- DL3006: Always tag the version of an image explicitly. Using untagged images can lead to non-reproducible builds.
- DL3007: Using the :latest tag in the FROM instruction. This is discouraged because the "latest" image changes over time, making builds unstable.
- DL3008: Pin versions in apt-get install. Failing to do so can result in different package versions being installed during different build cycles.
- DL3009: Delete the apt-get lists after installing packages. This is a best practice to reduce the final image size.
- DL3020: Use COPY instead of ADD for simply moving local files into the container. ADD has additional complex behaviors (like auto-extraction) that are often unnecessary and risky.
- DL4000: Mandatory MAINTAINER or LABEL for authorship. This ensures that the image creator is identifiable for support and security updates