

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

(MIT Thesis Viva Voice Examination)



Tribhuvan University

Institute of Science and Technology

Central Department of Computer Science & Information Technology

(CDCSIT TU)

Presented By:

Deepak Aryal

Roll no: 7925005

CDCSIT TU

Supervised By:

Asst. Prof. Bikash Balami

CDCSIT TU

Date: January 23, 2026

ABSTRACT

- This thesis explores automated detection of Dockerfile misconfigurations using transformer-based NLP models.
- A two-stage framework is proposed for binary and rule-specific misconfiguration classification.
- The models are evaluated on a dataset of public Dockerfiles.
- Results show high detection accuracy, with CodeBERT slightly outperforming RoBERTa.

Outline

- Introduction
- Problem Statement
- Objectives
- Scope and Limitation
- Literature Review
- Methodology
- Dataset Description
- Data Preprocessing
- Description of Algorithms
- Implementation Details
- Result Analysis
- Inference
- Conclusion
- Future Recommendation
- References

Introduction

- Docker is a core technology in modern DevOps for building and deploying applications
- Dockerfiles define container build processes but are complex and under-analyzed
- Misconfigurations in Dockerfiles can cause security vulnerabilities and build failures
- Existing tools detect rule violations but fail to generalize to complex patterns
- This research evaluates transformer-based NLP models to improve Dockerfile misconfiguration detection

This naturally leads to the problem addressed in this thesis.

Problem Statement

- Public Dockerfiles frequently violate best practices and contain security risks
- Many engineers lack deep security expertise when writing Dockerfiles
- Existing tools are rule based and limited to predefined checks which fail to detect unseen patterns
- NLP/AI for Dockerfiles is still underexplored

Problem Statement

traefik:latest [v2.11.34]

- Traefik http router rule



```
traefik.http.routers.${PROJECT}.rule=Host(`api.
${DOMAIN}`) ||
(HostRegexp(`{subdomain:(app)}.${DOMAIN}`
) && PathPrefix(`/api/`))
```

traefik:latest [v3.6.6]

- Traefik http router rule



```
traefik.http.routers.${PROJECT}.rule=Host(`api.
${DOMAIN}`) ||
(HostRegexp(`{subdomain:(app)}.${DOMAIN}`
) && PathPrefix(`/api/`))
```



```
traefik.http.routers.${PROJECT}.rule=Host(`a
pi.${DOMAIN}`) || ((Host(`
app.${DOMAIN}`) && PathPrefix(`/api/`))
```

Objectives

- 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

Scope and Limitation

- ⚡ **Models compared:** RoBERTa & CodeBERT
- 🔧 **Objective:** Automated detection of Dockerfile misconfigurations
- 📄 **Analysis type:** Static, NLP-based evaluation
- 📊 **Performance metrics:** Accuracy | Precision | Recall | F1-score | AUC-ROC

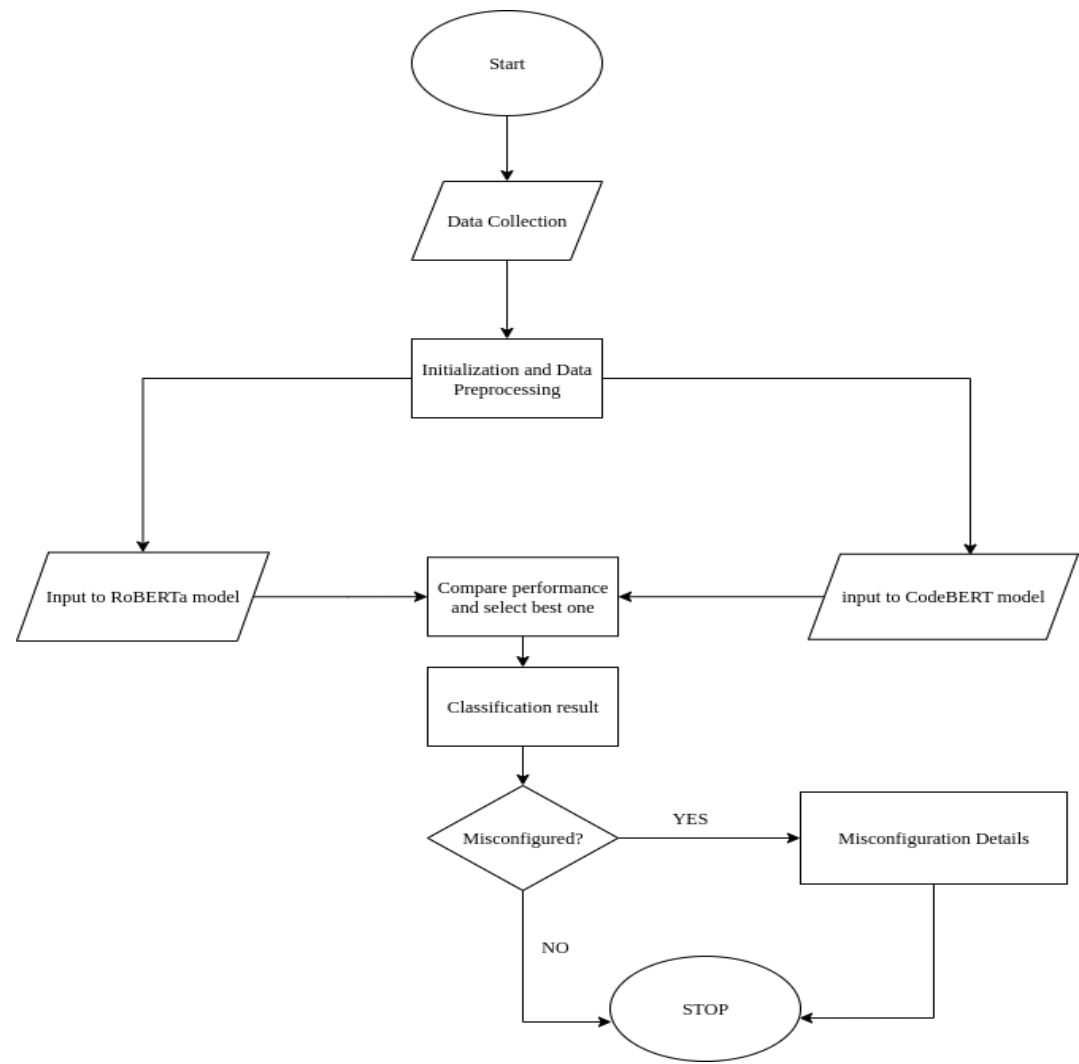
- 🐳 **Scope:** Only Dockerfiles
- 📁 **Dataset:** Publicly available Dockerfiles
- ⚡ **Models:** Only RoBERTa and CodeBERT
- 💻 **Resources:** Limited computational power may affect experiments and tuning

Literature Review

Study	Dataset	Key findings	Research gap
Learning from, DevOps Artifacts	Expert & non-expert Dockerfiles	Non-expert Dockerfiles violate best practices $\sim 6\times$ more; IDE integration suggested	Needs adaptive AI methods for complex misconfigs
Shipwright	Faulty Dockerfiles	Automated repair suggestions; Improves build correctness	Focus on build failure, not misconfiguration
Dockerfile Flakiness	Flaky Dockerfiles	Characterizes flakiness; Improves repair success	No line-level security detection
Dockerfile Linting	Docker Workflows	Ensures consistent quality via linting	No AI/NLP misconfiguration detection
Not all Dockerfile Smells are the Same	39,000 Dockerfiles	Experts overlook smells; Validates linting	No AI/NLP detection explored
DRIVE: Rule Mining	Dockerfiles	Identifies semantic & syntactic rules; Improves quality	Focus on static rules, not AI/NLP

Methodology

System flow chart



Dataset Description

- Source: Publicly available Dockerfiles from GitHub
- Primary Corpus: Henkel et al. Dockerfile corpus (ICSE 2020)
 - ~178,000 real-world Dockerfiles
- Extended data: Additional Dockerfiles collected via GitHub Search API
- Processing and labels: Dockerfiles parsed line by line; each instruction (e.g., FROM, RUN, COPY) labeled as correct or wrong for training and evaluation of NLP models.
- Dataset Size:
 - ~178,800 Dockerfiles
 - ~1.6 million Dockerfile instructions

Data Preprocessing

Model training strategy

- Stage 1: Binary misconfiguration classification
 - Correct vs wrong
- Stage 2: Rule-specific misconfiguration classification
 - Misconfigured instructions with rule id

Data Preprocessing

Data splitting

- Stratified splitting into:
 - Training: 80%
 - Validation: 10%
 - Test: 10%
- Separate strategies for stage 1 and stage 2
- Due to computational resource constraints, a representative subset of the dataset was used for training and evaluation.

Data Preprocessing

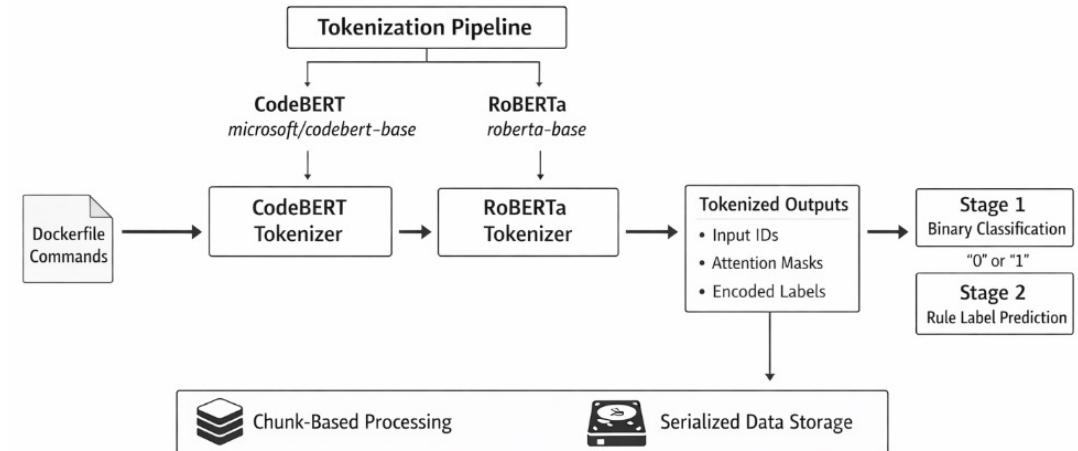
Data splitting

Aspect	Stage 1: Binary classification	Stage 2: Rule-level classification
Task	Correct vs Wrong	Rule-level misconfiguration classification
Initial data	~1.6M instructions	Misconfigured instructions only
Data reduction and balancing method	Downsampled and selected 200K samples per class	Rare rules removed (<200 samples) and capped dominant rules (<=5K)
Final split	80% / 10% / 10% (stratified)	80% / 10% / 10% (rule-stratified)
Motivation	Computational feasibility	Fair multi-class learning and fair computational feasibility

Data Preprocessing

Tokenization and encoding

- Unified subword tokenization applied for both learning stages
- Fixed input length of 256 tokens with truncation and padding
- Training set only label encoding, with pre-tokenized inputs stored for efficient reuse

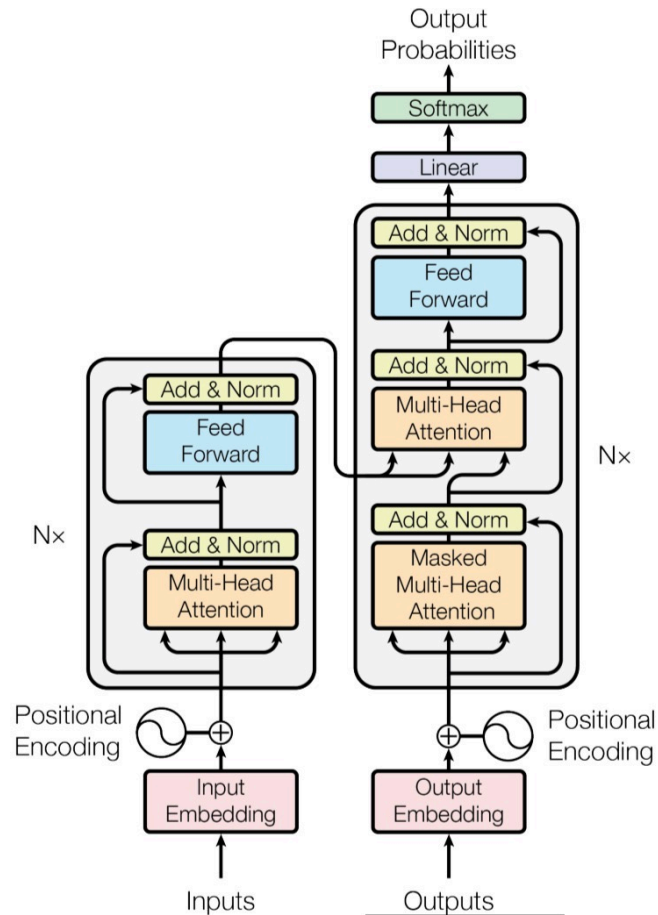


Data Preprocessing

Dataset summary

Aspect	Stage 1: Binary classification	Stage 2: Rule-level classification
Initial dataset	~1.6M instructions	Same dataset
Classes	2 (correct vs wrong)	63 rules
Class balance	2,00,000 per class	Filtered 200+ samples, capped at 5,000 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
Purpose	Detect misconfiguration	Predict violated rule if misconfigured

Description of Algorithms



- Faster than RNNs/LSTMs: parallel training on large datasets
- Attention mechanism: focuses on most relevant words in context
- Encoder/Decoder: encoder for classification, both for translation
- Self-attention: captures dependencies even across distant words
- Positional encoding: keeps track of word order

Description of Algorithms

RoBERTa and CodeBERT

- Encoder only transformer architecture
- Optimized version of BERT with larger datasets and better training strategies
- Dynamic masking: different tokens are masked in each epoch

Description of algorithms

RoBERTa and CodeBERT

Transformer encoder

- Stack of self-attention layers producing contextualized token representations.
- Input tokens => embedding vectors => processed by encoder layers.

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

- Where Q, K, V: query, key, value from embeddings and d_k : dimension of key vectors

Classification head

- The [CLS] token embedding from the encoder is passed through a feed-forward layer to produce a probability distribution over class labels, enabling classification of Dockerfiles as correct or misconfigured.

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

- Where W, b = trainable parameters and y = probability distribution over classes

Implementation Details

Model architecture and input processing

Pretrained models

- RoBERTa-base (text-oriented)
- CodeBERT-base (code-aware)

Tokenization

- Model-specific tokenizers
- Subword encoding
- Maximum sequence length: 256 tokens

Input granularity

- Dockerfile commands processed at line level

Output heads

- Binary classifier (Stage 1)
- Multi-class classifier (Stage 2)

Implementation Details

Training configuration and optimization

Optimizer: AdamW

Learning rate:

- Binary classification: $2 * 10^{-5}$
- Rule classification: $5 * 10^{-5}$

Batch size: 32

Epochs: 3

Training strategy:

- Validation-based loss monitoring
- Gradient accumulation

Loss function:

- Binary classification: Cross-entropy loss
- Rule classification: Multi-class cross-entropy loss

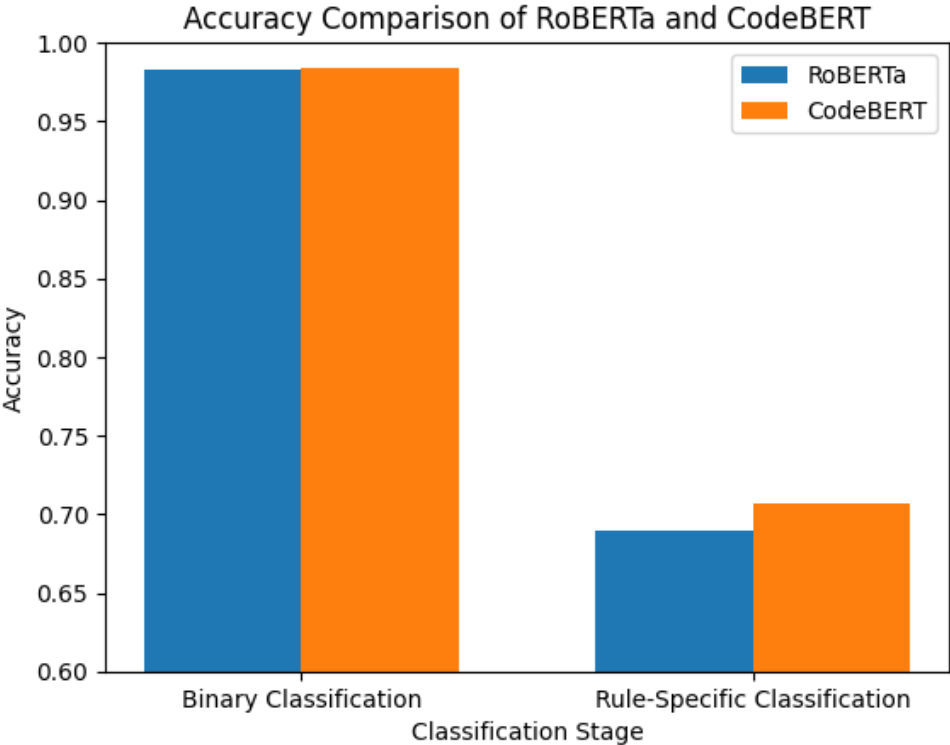
Model persistence:

- Epoch-wise checkpointing for reproducibility

Result Analysis

Accuracy

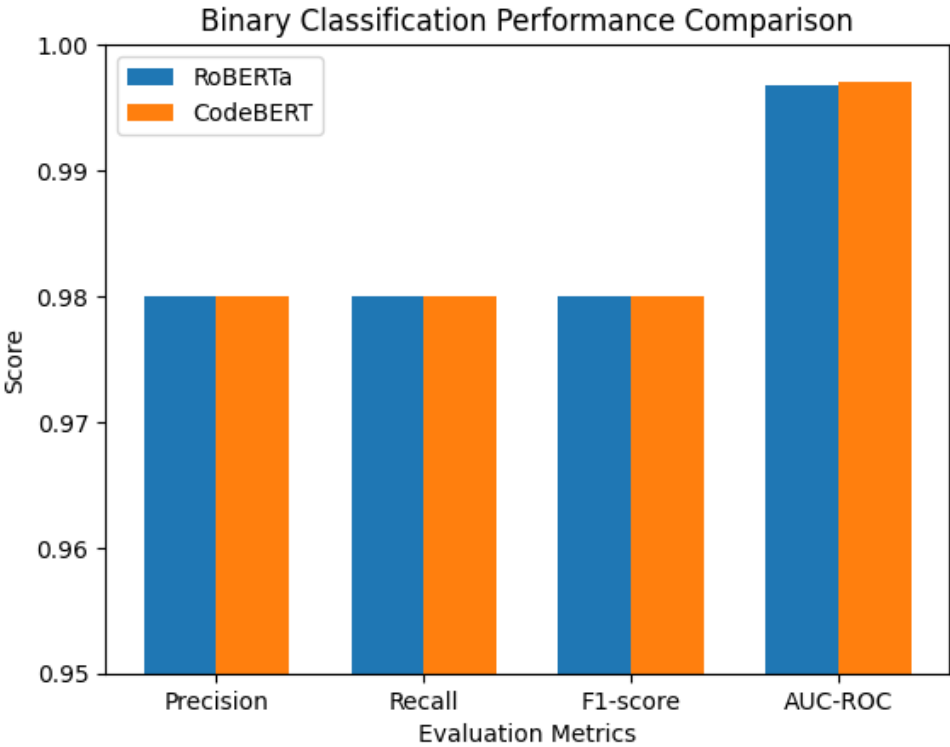
Stages	Models	RoBERTa	CodeBERT
Binary classification		0.98	0.98
Multi class classification		0.7048	0.7070



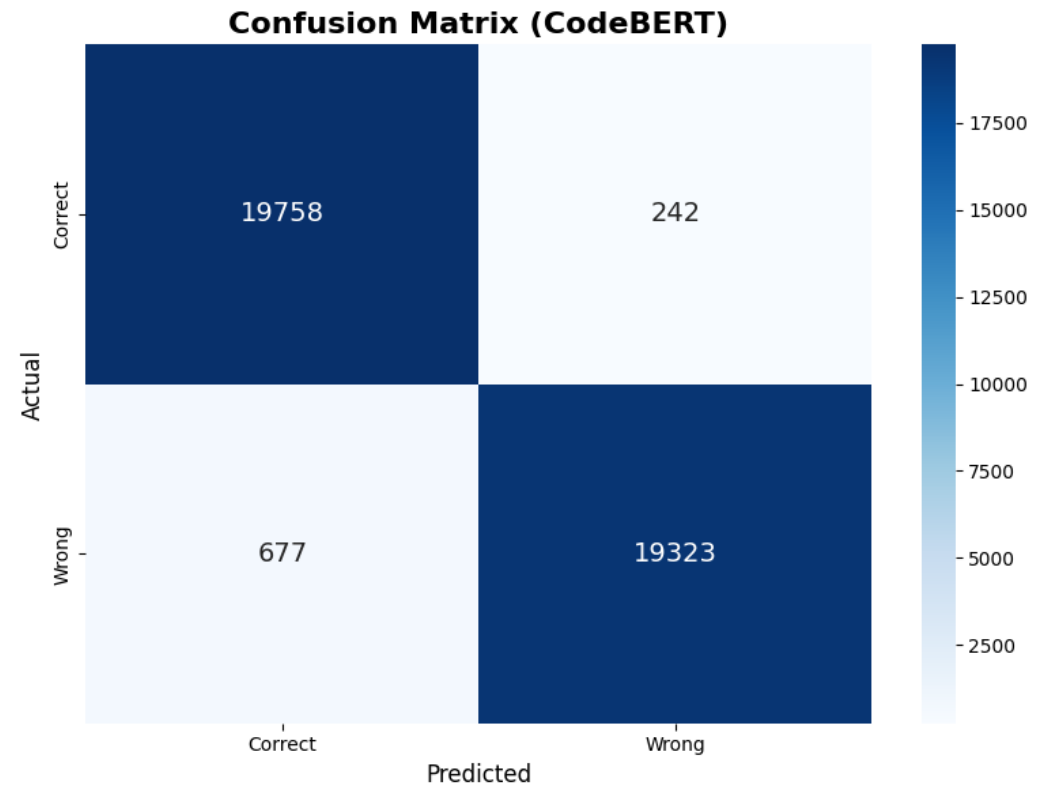
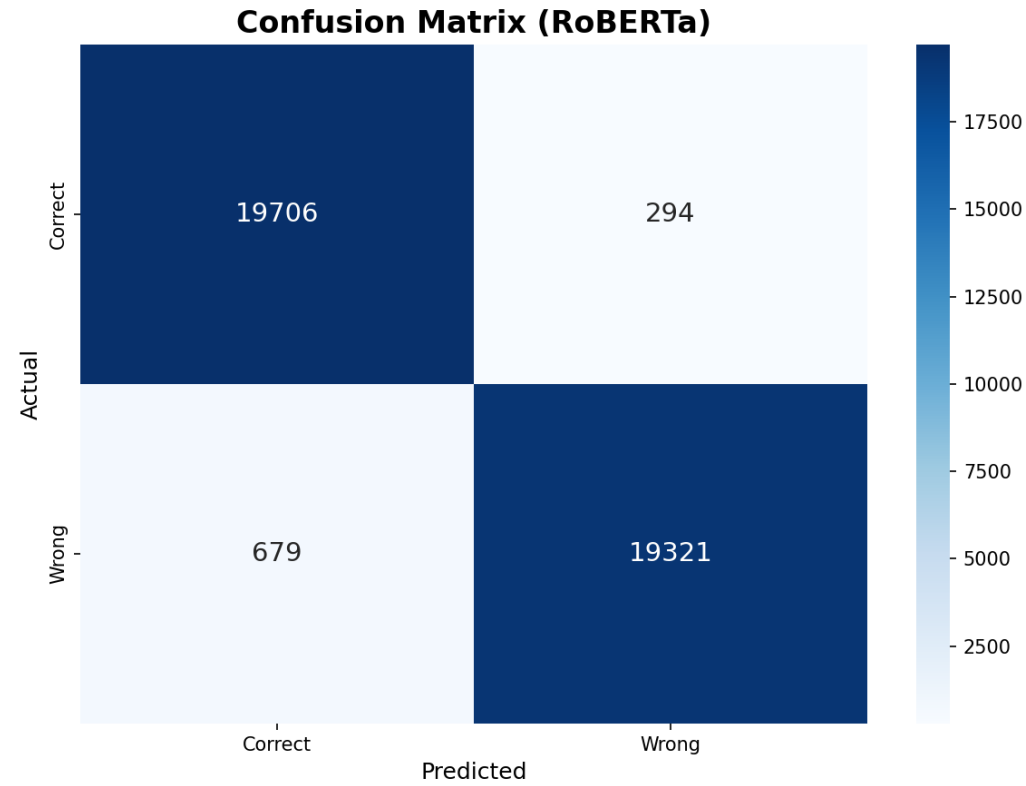
Result Analysis

Binary classification performance

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



Result Analysis



Result Analysis

Rule(Multi) classification performance (Overall)

<div>Models</div> <div>Metrics</div>	RoBERTa	CodeBERT
Macro-averaged F1	0.7055	0.7136
Weighted-averaged F1	0.7013	0.7040
Top-1 accuracy	0.7048	0.7070
Top-3 accuracy	0.8901	0.8938

Result Analysis

Per class rule(multi) classification performance


<div>Models</div> <div>Metrics</div>	RoBERTa	CodeBERT
Total number of rules	63	63
Average F1-score	0.7055	0.7136
Number of rules with $F1 > 0.8$	21	21
Number of rules with $F1 < 0.5$	11	11

Inference

RoBERTa

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


=====
DOCKERTFILE 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)
-----
```

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

=====
DOCKERTFILE 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
-----
```

Inference

CodeBERT

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

=====
DOCKERTFILE 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)
=====
```

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

=====
DOCKERTFILE 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
=====
```

Conclusion

- For binary misconfiguration detection, both CodeBERT and RoBERTa achieve excellent and identical performance ($\approx 98\%$ accuracy), demonstrating strong and reliable classification capability.
- In rule-specific misconfiguration classification, CodeBERT performs slightly better than RoBERTa, indicating its stronger ability to capture Dockerfile-specific and semantic patterns.
- Model selection depends on application needs: both models are suitable for large-scale binary detection, while CodeBERT is more effective for detailed rule-level misconfiguration analysis.

Future Recommendation

- Improve rule-specific classification using larger and more diverse datasets, including data augmentation for rare rules
- Explore ensemble models or hybrid approaches combining transformers with rule-based heuristics
- Integrate the proposed models into CI/CD pipelines for real-time misconfiguration detection
- Extend the approach to other IaC technologies such as Terraform, CloudFormation, and Ansible

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>

References

- [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>

References

- [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.

Thank you!!!