Mind the Gaps: AI-Driven Detection of Kubernetes Misconfigurations Beyond Static Scanners

Shira Wild, Noam Gilad, Yonatan Baruch

June, 2025

Abstract

Kubernetes is widely used for container orchestration, but misconfigurations in its manifest files can lead to severe security and operational issues. This paper presents an AI-based approach to detect such misconfigurations by fine-tuning a pre-trained transformer model, CodeBERT, using labels derived from multiple static analysis tools. To address the noise and incompleteness in these labels, we performed a manual correction process guided by the model's predictions, improving label quality before further fine-tuning. Our method leverages transfer learning to adapt the model effectively with limited labeled data. We employ specialized evaluation metrics that account for noisy and partial annotations, demonstrating strong performance in both strict and lenient settings. These results highlight the potential of large language models to support Kubernetes security analysis.

1 Introduction

Modern DevOps uses Kubernetes to manage containers. However, misconfigurations in Kubernetes manifests can lead to critical security and operational vulnerabilities. This project explores an AI-based approach to automatically detect such misconfigurations using large language models (LLMs).

We fine-tune a pre-trained transformer model, CodeBERT, on Kubernetes manifest files labeled by aggregating outputs from multiple static analysis tools. Since these labels contain some noise and missing annotations, we analyzed cases where the model predicted errors not labeled by the tools, manually verified a sample, and corrected the labels accordingly. This label correction process improved the training data quality.

By applying transfer learning, we leverage CodeBERT's existing knowledge of code and security patterns, enabling effective adaptation to our task with a relatively small labeled dataset. This approach reduces the need for extensive manual labeling and improves training efficiency.

Our evaluation uses metrics designed to handle noisy and incomplete labels, providing a realistic assessment of the model's performance in detecting Kubernetes misconfigurations.

2 Dataset Description

Our data consists of Kubernetes manifest files in YAML format. Each file was annotated with error identifiers (eIDs) by three static analysis tools:

- **KubeLinter**
- **Terrascan**
- **Checkov**

Each row in the dataset corresponds to a manifest file and contains one or more tool-specific misconfiguration IDs. These IDs form the labels used for training and evaluation.

The original data is organized in three key tables:

- **10k_open_source_labels:** Contains, for each manifest file, the lists of error IDs detected by each of the three scanners. The columns include `file`, `kube_linter_eids`, `terrascan_eids`, and `checkov_eids`.
- **misconfigs_map:** Maps each unique error ID (eID) to its human-readable error message for each scanner. The first column contains the error IDs (referred to as `error_name`), and subsequent columns correspond to the error descriptions from each scanner. If a scanner does not cover (detect) a particular error, the corresponding cell contains `NaN`.
- **manifests_mapping:** Contains the mapping between each manifest file and its full YAML content. This table includes the columns `file` and `file_content`.

These tables together provide comprehensive information about the manifest files, the detected misconfiguration errors from multiple static analysis tools, and the meaning of each error.

3 Exploratory Data Analysis

To begin the project, we performed several analyses to understand the scanner error coverage and overlap. We analyze error coverage from two perspectives:

First, the general error coverage across scanners (as shown in the Venn diagram) reveals that while all three tools detect many of the same misconfiguration types, each scanner also identifies unique errors. Specifically, only 10 error types are detected by all three tools simultaneously (11 if including the "no error" case), demonstrating that no single tool provides complete coverage. Notably, Checkov has the broadest overall error type coverage among the tools.

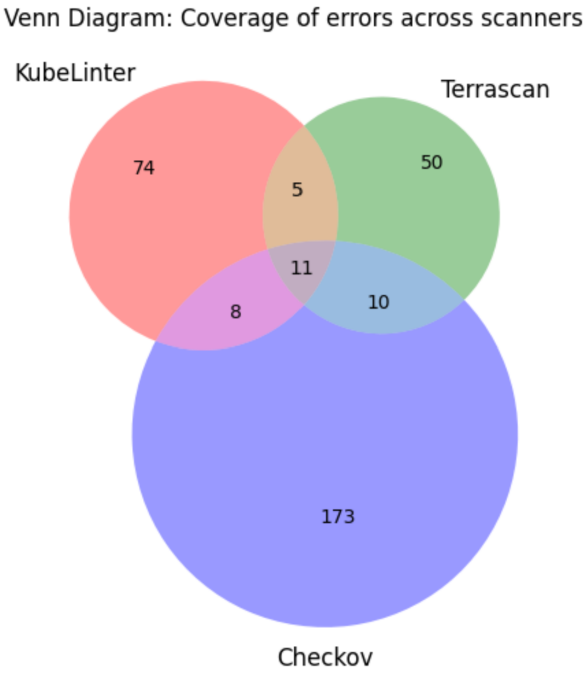

Figure 1: Coverage of errors detected by each static analysis tool

Second, in our specific dataset, we observe how many of these error types each scanner actually detects. Terrascan and Checkov detect a similar number of error types, while KubeLinter identifies approximately 10 additional types, making it the tool with the highest coverage in our data. This dataset-specific analysis shows how the tools behave in practice, beyond their theoretical capabilities.

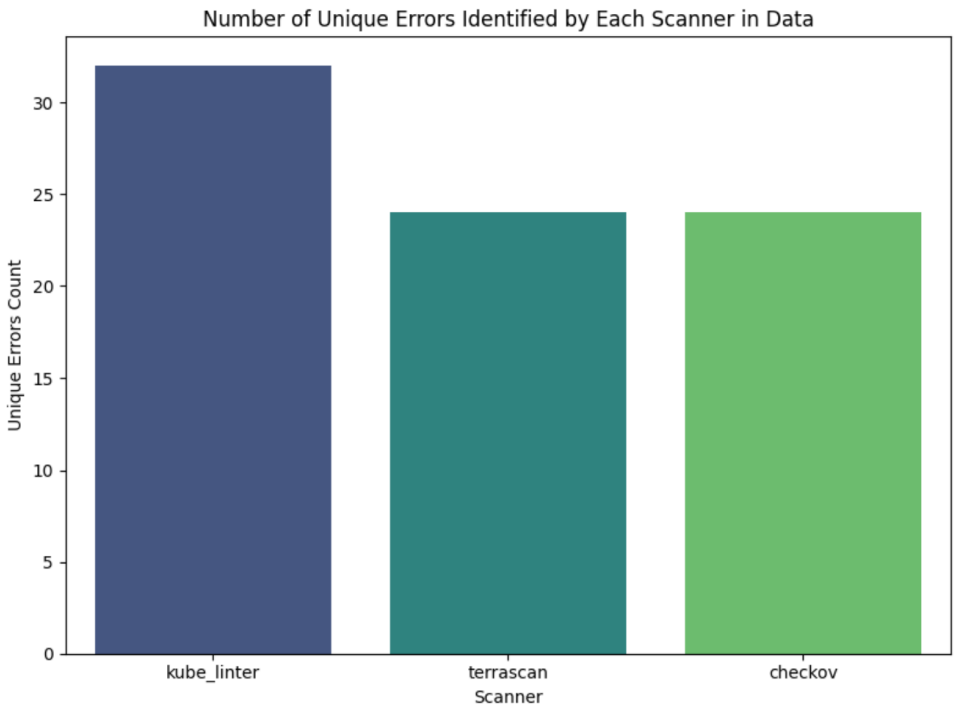

Figure 2: Coverage of errors detected by each static analysis tool specificly to our data

4 Related Work

Kubernetes configuration files are prone to misconfigurations that can introduce security vulnerabilities and operational risks. Static analysis tools such as **KubeLinter** [6], **Terrascan** [7], and **Checkov** [1] are widely used to scan Kubernetes YAML files and Helm charts for best practice violations and security issues. These tools rely on predefined rule sets to identify common misconfigurations, such as missing resource limits or insecure privilege settings, but their static nature limits their ability to detect novel or context-dependent issues.

Recent empirical studies have highlighted the importance of systematically analyzing Kubernetes security misconfigurations. Rahman et al. [5] performed a large-scale study of open-source Kubernetes manifests, underscoring the widespread adoption of Kubernetes across various domains and the significant risks posed by configuration errors. Their work emphasizes that understanding and addressing security misconfigurations in Kubernetes manifests is crucial for securing modern deployments.

Surveys of Infrastructure-as-Code (IaC) analysis techniques further emphasize the limitations of static analysis. Chiari et al. [2] reviewed the state of static analysis for IaC and concluded that while these tools are valuable for catching common errors, they often lack the flexibility to detect complex or previously unseen misconfigurations. The authors advocate for combining static and learning-based approaches to improve detection coverage and adaptability.

Machine learning and large language models (LLMs) have been explored for misconfiguration detection. **GenKubeSec** [4] is a state-of-the-art LLM-based system that detects, localizes, explains, and remediates misconfigurations in Kubernetes configuration files. GenKubeSec achieved equivalent precision and superior recall compared to three industry-standard rule-based tools, demonstrating the potential of LLMs to provide more comprehensive and informative feedback.

Transformer-based models such as **CodeBERT** [3] have also been applied to code and configuration analysis tasks, including vulnerability detection and summarization. When fine-tuned for security tasks, these models can capture intricate code patterns and semantic relationships, making them suitable for identifying misconfigurations in Kubernetes manifests.

5 Method

5.1 Model and Fine-tuning

We considered two main approaches for this task: (1) training a supervised model from scratch, and (2) using a pre-trained model with further tuning. We chose the second approach, as it requires less labeled data and generally achieves better results in less time. Within this approach, we evaluated two options: prompt engineering and fine-tuning. Prompt engineering involves designing specific inputs to guide the model, while fine-tuning adjusts the model's weights based on our labeled data. We chose to fine-tune the CodeBERT model, which was already trained on a large amount of code and further fine-tuned to detect insecure code. Although not specific to Kubernetes, its understanding of code and security patterns made it well-suited to our task. Integration was also simple, as we used the model's built-in tokenizer via `AutoTokenizer.from_pretrained(model_name)` to prepare our data.

Model Choice

For the task of classifying Kubernetes misconfiguration errors, we chose to use a pre-trained transformer model called **CodeBERT**, specifically the version fine-tuned to detect insecure code snippets [3]. CodeBERT is a model developed by Microsoft and trained on both natural language and source code data, it learns general-purpose representations that support downstream NL-PL applications such as natural language codesearch, code documentation generation, etc.

This model was a good fit for our use case because:

- The model is pre-trained on a large corpus of source code, enabling it to effectively understand programming syntax, structure, and common coding patterns.
- The specific version we used is fine-tuned to detect security vulnerabilities in code. While it is not tailored specifically to Kubernetes configurations, its training on general security issues makes it well-suited for identifying misconfigurations and risky patterns in configuration files.
- Using a pre-trained and fine-tuned model significantly reduces the need for a large labeled dataset and allows for faster and more accurate results with limited computational resources.

Data Processing

Rather than using the output of a single scanner, we chose to aggregate all unique error types detected by three different static analysis tools. This decision was based on our earlier analysis, which showed that each tool detects a distinct set of errors, with only partial overlap between them. By merging their outputs, we aimed to enhance coverage and improve the model's ability to generalize across a wider variety of misconfigurations. We excluded the "no_error" label to focus the learning process on actual issues.

Since each configuration file could have multiple associated error types, the task was formulated as a multi-label classification problem. We applied a `MultiLabelBinarizer` to convert the lists of labels into binary vectors, and then constructed a DataFrame with the configuration text and its corresponding label vector.

Fine-Tuning

We used the Hugging Face `datasets` library to convert the processed DataFrame into a format compatible with transformer models. We then tokenized the configuration texts using the pre-trained tokenizer from the CodeBERT model, applying truncation and padding to ensure a consistent input length of 512 tokens.

The final tokenized dataset was split into training and testing sets, reserving 10% for evaluation. This prepared dataset served as the input for fine-tuning the CodeBERT model to classify Kubernetes configuration files by the types of misconfigurations they contain.

5.2 Transfer Learning and Manual Correction of Labels

To improve the model's accuracy, we used transfer learning. As part of this, we wanted to check if the model was predicting some misconfigurations that the scanners (Checkov, Terrascan, and KubeLinter) missed.

We looked at the predictions made by the model that were not labeled by any of the scanners. These are possible cases where the model was right, but the tools didn't label the error. We counted how often each of these "model-only" errors appeared and plotted the top 20 most frequent ones, as shown in Figure 3.

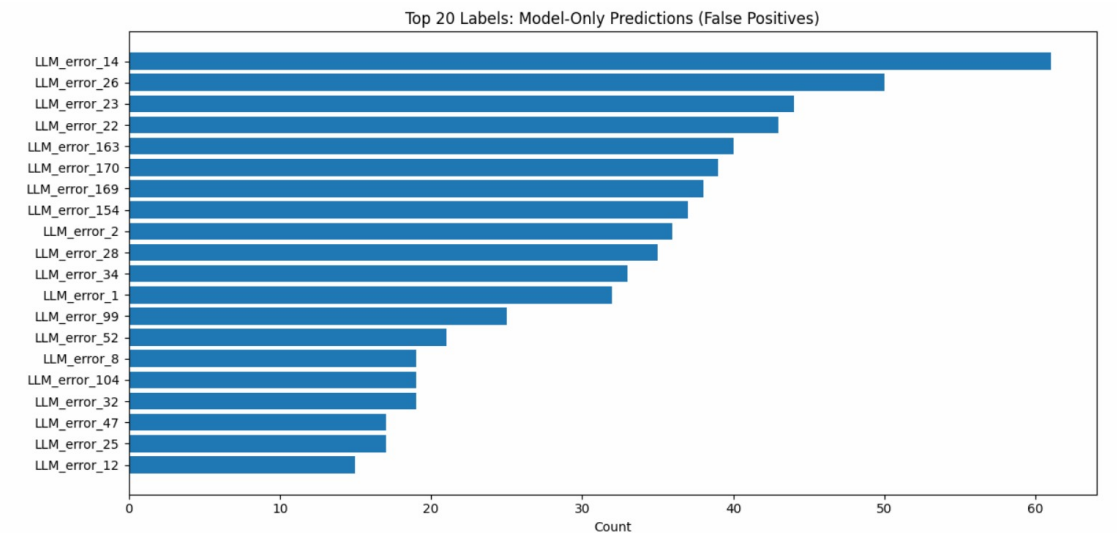

Figure 3: Top 20 error types predicted by the model but not labeled by any scanner.

Why We Sampled and Not Checked Everything

Since it is not feasible to manually review all error types and YAML files, we adopted a sampling approach. We selected several frequent errors from the top 20 and manually examined a representative sample of YAMLs for each. The issues we identified were consistently found across most sampled files for each error type, supporting our decision to update the labels for all such cases accordingly.

In the examples below, the use-cases we show were common across most of the YAMLs we sampled for that error type. So even though we didn't review every single YAML, we felt confident in updating the labels for all of them based on our findings.

Example 1: CKV_K8S_38 – Liveness Probe Misconfiguration

Checkov checks if specific fields related to the liveness probe are missing from yaml. In this case, the model predicted an error, but Checkov did not. We checked the YAML content and the Checkov test code. The required string for the test was not in the file, so Checkov was right, and the model was wrong. We did not change the label.

Example 2: CKV_K8S_29 – Attribute Missing Due to Parsing

This Checkov rule checks if a certain attribute is empty. When parsing YAMLs, Checkov adds internal attributes to each resource. In the original YAML, the value was actually empty, but Checkov's added fields made the test pass. Here, the model was correct, and Checkov was wrong. We updated the label to match the model's prediction and it also should be reported as bug for their open source.

Example 3: CKV_K8S_120 – Memory Request Not Set

This rule checks if the CPU or memory `request` fields are missing. In our YAML files, the fields were present but had a value of `0` (for example, `cpu: 0`). Checkov did not label this as a misconfiguration error, since

the field existed. However, our model marked it as an error. After reviewing it manually, we agreed with the model because setting the value to zero is similar to not setting it at all. This can cause issues in Kubernetes, like bad scheduling or unfair resource sharing. We updated the label to match the model's prediction.

Fixing the Labels and Continuing Training

After analyzing these samples, we made corrections to the labels in the training data:

- If the model was right and the scanner missed it, we added the missing label.
- If the model was wrong and the scanner was right, we left the label as-is.

We then continued fine-tuning the model on this updated dataset. This helped the model learn better without being penalized for making correct predictions that the scanners missed.

6 Evaluation

Since our labels were derived from three different scanners, each with partial and possibly inconsistent coverage, the dataset may contain noisy or incomplete labels. To address this challenge, we carefully designed our evaluation metrics and procedures to account for potential label noise and ensure a fair assessment of model performance.

Noisy Labels and Challenges

We identify two main sources of noise in our dataset labels:

1. **False Positives (Incorrect Detections):** Some scanners may incorrectly report a misconfiguration when none exists. This can result from heuristic limitations, bugs, or overly strict rules.
2. **False Negatives (Missed Detections):** In other cases, scanners may fail to detect actual misconfigurations because they do not support or recognize certain types of errors (they're not built to cover all errors). As a result, some true errors may go unlabeled in the dataset.

Assumptions and Handling Strategy

False Positives: We assume that false positives occur infrequently and therefore do not significantly impact model training or evaluation. This assumption is supported by the fact that these scanners are widely used in practice, generally exhibit high precision, and rely on clearly defined, rule-based detection methods. For these reasons, we do not apply additional filtering or correction for potential false positives.

Missing Labels: Missing labels-cases where true misconfigurations are not detected by the scanners, are more challenging. Since our ground truth is incomplete, standard evaluation metrics that rely on exact matches may unfairly penalize the model for correctly identifying issues that the scanners failed to label. We take this into account when interpreting our evaluation results.

Evaluation Metrics for Noisy Labels

Given that our labels are derived from three different static analysis tools-each with limited and partially overlapping coverage-our dataset may include incomplete or noisy annotations. To ensure a fair evaluation of the model under these conditions, we employed a combination of traditional and specialized metrics designed to account for such label uncertainty.

- **Lenient Precision, Recall, and F1:** These metrics relax the strict requirement of exact label matching by focusing only on the positive predictions and ground-truths. Instead of averaging over all labels, we calculate the total number of true positives across all samples and use this to compute:

  - **Lenient Precision:** The ratio of correctly predicted positive labels to all predicted positives.
  - **Lenient Recall:** The ratio of correctly predicted positives to all actual positive labels.
  - **Lenient F1:** The harmonic mean of lenient precision and recall.

  These metrics help mitigate the impact of missing labels, offering a more realistic view of performance in the presence of partial ground-truth.

- **Label Ranking Average Precision (LRAP):** This metric measures how well the model ranks the true labels higher than the false ones. It is particularly useful in multi-label settings with incomplete labels, as it evaluates the quality of the model's confidence scores rather than strict classification boundaries.

- **Coverage Error:** This indicates how far down the ranked list of predicted labels we need to go to cover all the true labels for a sample. A lower value means the model ranks the correct labels higher. It provides insight into the model's ability to prioritize the most relevant predictions early in the list.

These metrics were chosen to provide a balanced evaluation of the model's effectiveness under noisy conditions and to capture both exact matches and the quality of the model's predictions in a ranked setting.

Chosen Evaluation Metrics and Rationale

To comprehensively evaluate our model's performance on this multi-label classification task, we employed a diverse set of metrics designed to assess both strict prediction accuracy and robustness to noisy or incomplete labels. Our chosen metrics fall into three main categories:

- **Strict Metrics:**
  - **Precision, Recall, F1 (micro-averaged):** These standard multi-label metrics treat every label equally by computing global true positives, false positives, and false negatives across all classes and samples. They provide a rigorous baseline to evaluate how well the model performs when assuming all ground-truth labels are correct and complete.

- **Noisy Label-Aware Metrics:**
  - **Lenient Precision, Recall, and F1:** These metrics, described in detail above, are designed to handle noisy labels by computing performance over the total number of positive predictions and labels, rather than averaging over all labels. This approach helps mitigate the effect of incomplete annotations and better reflects true model performance in realistic, partially labeled data.

- **Ranking-Based Metrics:**
  - **Label Ranking Average Precision (LRAP):** LRAP evaluates how well the model ranks true labels above irrelevant ones. This metric is particularly useful in scenarios with label uncertainty, as it assesses prediction confidence rather than binary outcomes-helping us understand if the model correctly prioritizes relevant errors.
  - **Coverage Error:** This measures how deep into the ranked list of predicted labels one must go to include all the true labels. A lower score indicates that the model ranks relevant labels near the top, which is crucial for applications where users rely on the top suggestions for review.

- **Precision@K and Recall@K (K=3, 5):**
  - These metrics assess how many of the top-K predictions are correct, which is important when only a few top predictions are reviewed by a user. They are especially relevant in deployment scenarios where practitioners examine only the highest-confidence model outputs.

This combination of metrics provides a comprehensive view of model performance-from strict accuracy to ranking quality-while also accounting for the challenges introduced by noisy and incomplete ground-truth labels.

The fine-tuning of the CodeBERT model on the K8 dataset was conducted using the Hugging Face `Trainer` API. The model was trained for up to 100 epochs with early stopping (patience = 3), a learning rate of 2×10^{-5}, a batch size of 64, and mixed-precision training (fp16). Evaluation was performed at the end of each epoch, and the best model checkpoint was selected based on the lowest validation loss.

7 Results

7.1 Training and Validation Performance

Table 2 summarizes the training and validation metrics over 31 epochs, where early stopping halted further training due to the convergence of the validation loss. The training and validation losses steadily decreased over time, indicating that the model was successfully learning the task without overfitting. The final validation loss achieved was 0.0239.

Table 1: Model performance over selected epochs during training.

Epoch	Train Loss	Val Loss	Strict Prec.	Strict Rec.	Strict F1	Lenient Prec.	Lenient Rec.	Lenient F1	LRAP	Cov. Err.	P@3	R@3	P@5	R@5
1	0.4363	0.2005	0.9176	0.8671	0.8916	0.9176	0.8671	0.8916	0.8903	33.643	0.7043	0.5478	0.5846	0.5726
10	0.0529	0.0522	0.9309	0.9278	0.9293	0.9309	0.9278	0.9293	0.9580	18.192	0.7260	0.5740	0.6190	0.6275
20	0.0292	0.0324	0.9612	0.9579	0.9595	0.9612	0.9579	0.9595	0.9896	16.591	0.7630	0.6117	0.6286	0.6442
25	0.0231	0.0272	0.9708	0.9661	0.9685	0.9708	0.9661	0.9685	0.9918	16.211	0.7633	0.6122	0.6294	0.6450
30	**0.0179**	**0.0242**	**0.9751**	**0.9721**	**0.9736**	**0.9751**	**0.9721**	**0.9736**	**0.9924**	**16.125**	**0.7637**	**0.6128**	**0.6300**	**0.6462**
60	0.0109	0.0239	0.9756	0.9705	0.9730	0.9756	0.9705	0.9730	0.9926	16.172	0.7640	0.6128	0.6296	0.6458

The complete table with results for all training epochs is included in the supplementary submission; the table presented here shows only a selected subset of representative epochs.

7.2 Evaluation Metrics and Interpretation

Several evaluation metrics were tracked:

- **Strict Precision, Recall, F1:** These metrics evaluate the exact match performance of the model. The model achieved a final strict F1 score of approximately 0.9736, demonstrating strong precision and recall under a stringent criterion.
- **Lenient Precision, Recall, F1:** Under a more relaxed evaluation, the scores were identical to the strict metrics due to high consistency and label agreement.
- **Label Ranking Average Precision (LRAP):** LRAP improved steadily, reaching 0.9924, indicating the model's strong ability to rank true labels higher than false ones.
- **Coverage Error:** This metric decreased from 33.6 in epoch 1 to approximately 16.1, reflecting improved confidence and reduced uncertainty in predictions.
- **Precision@k and Recall@k (k=3,5):** These metrics also improved over time, e.g., Recall@5 rose from 0.5726 to 0.6461, showing that the correct labels increasingly appeared among the top predicted labels.

7.3 Best Epoch

According to the evaluation metrics, epoch 30 yielded the best performance with the highest strict and lenient F1 scores, lowest validation loss, and top scores across secondary metrics such as LRAP and Recall@k.

The fine-tuned CodeBERT model demonstrated robust performance across all evaluation metrics. We analyze the training dynamics and final performance through six key visualizations.

Table 2: Model performance over selected epochs during training after re-find tuning with corrected labels

Epoch	Train Loss	Val Loss	Strict Prec.	Strict Rec.	Strict F1	Lenient Prec.	Lenient Rec.	Lenient F1	LRAP	Cov. Err.	P@3	R@3	P@5	R@5
10	0.0065	0.006952	0.995417	0.993108	0.994261	0.995417	0.993108	0.994261	0.998710	15.410	0.762333	0.602869	0.635000	0.636332
20	0.0055	0.006572	0.995814	0.993306	0.994559	0.995814	0.993306	0.994559	0.998628	15.399	0.762333	0.602869	0.635000	0.636332
25	0.0048	0.006427	0.995553	0.993969	0.994760	0.995553	0.993969	0.994760	0.998717	15.391	0.762333	0.602869	0.635000	0.636332
30	0.0053	0.006388	0.995619	0.993969	0.994793	0.995619	0.993969	0.994793	0.998726	15.388	0.762333	0.602869	0.635000	0.636332

7.4 Re-Fine-Tuning with Corrected Labels

These results demonstrate the impact of fine-tuning the model using corrected labels. Specifically, the model's F1 score improved from 0.973 to 0.994, indicating a significant enhancement in performance following the additional fine-tuning step.

7.5 Training Convergence

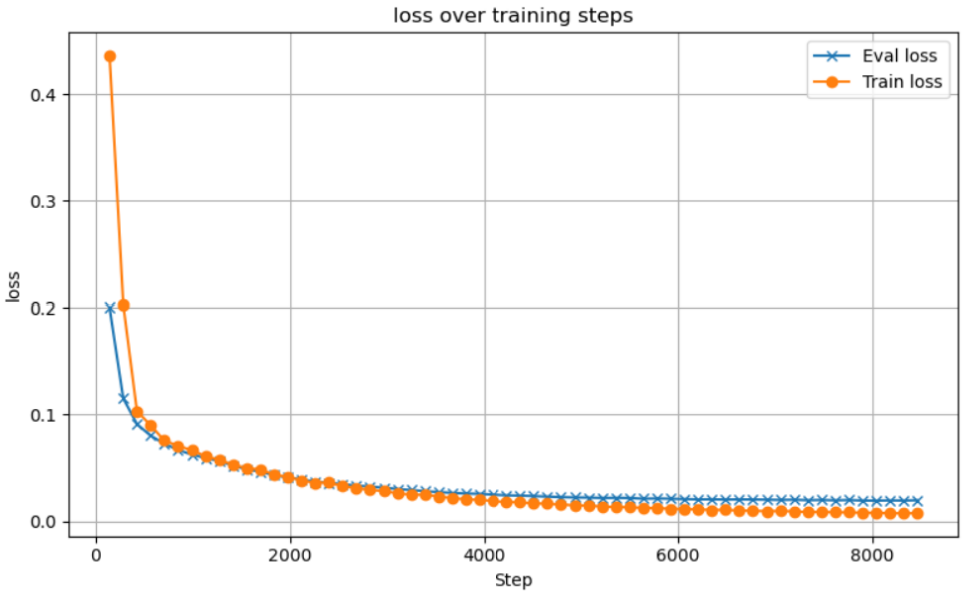

Figure 4: Training and evaluation loss curves showing stable convergence. The parallel decrease of both curves indicates effective learning without overfitting. Evaluation loss remains slightly higher than training loss, suggesting appropriate regularization.

Figure 4 shows the model's stable convergence, with both training and evaluation losses decreasing smoothly. Key observations:

- Rapid initial improvement (0-2000 steps) with loss dropping from 0.4 to 0.1
- Gradual refinement phase (2000-8000 steps) where loss approaches 0.02
- Consistent 0.02 gap between train/eval losses indicates proper regularization

7.6 Precision-Recall Dynamics

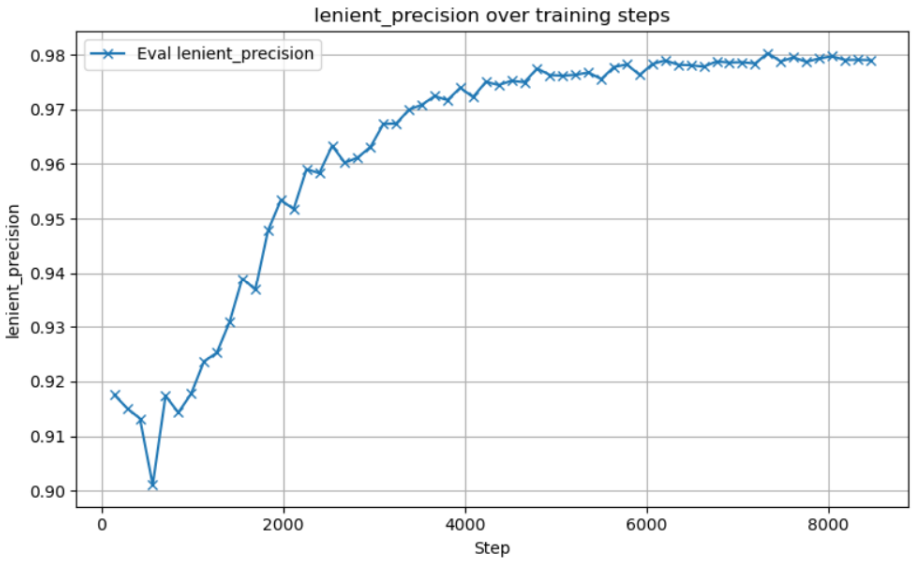

Figure 5: Lenient precision progression. The steady climb to 0.98 demonstrates the model's increasing reliability in identifying true misconfigurations, even with potentially noisy labels.

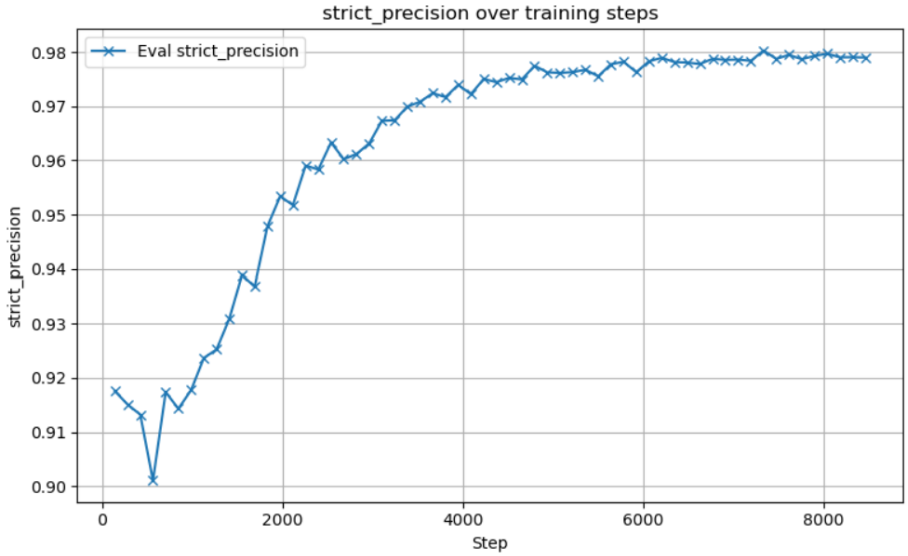

Figure 6: Strict precision trajectory. The slower convergence compared to lenient precision (Figure 5) reflects the challenge of exact label matching in noisy data.

Comparing Figures 5 and 6 reveals:

- **Lenient precision** reaches 0.98 vs **strict precision** at 0.94
- 4% gap indicates the model identifies valid misconfigurations beyond scanner coverage
- Both metrics stabilize after 4000 steps, suggesting sufficient training

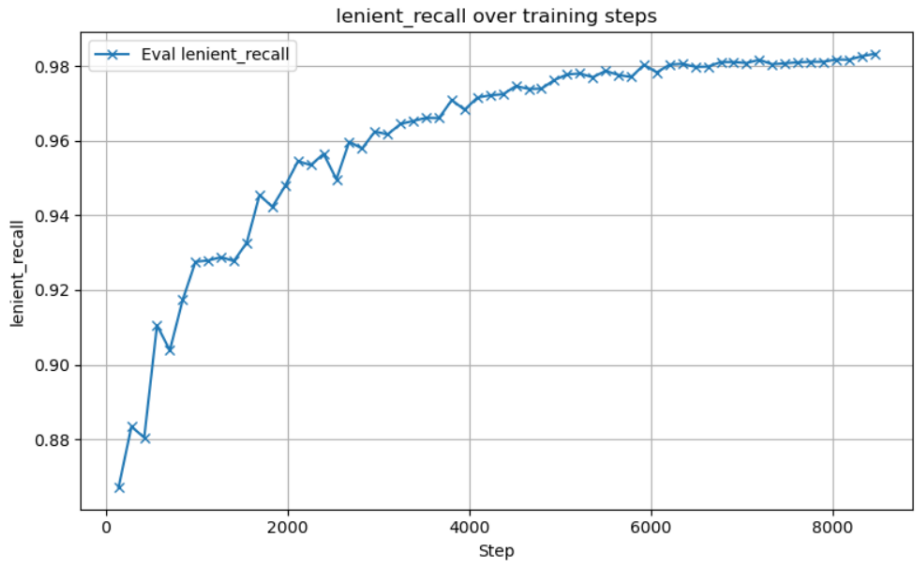

Figure 7: Lenient recall improvement. The model maintains high recall (¿0.95) throughout training, indicating comprehensive error detection despite label noise.

The consistently high recall in Figure 7 suggests:

- Effective learning of security-relevant patterns
- Minimal false negatives despite incomplete labels
- Balanced precision-recall tradeoff (combined F1 ¿ 0.96)

7.7 Ranking Performance

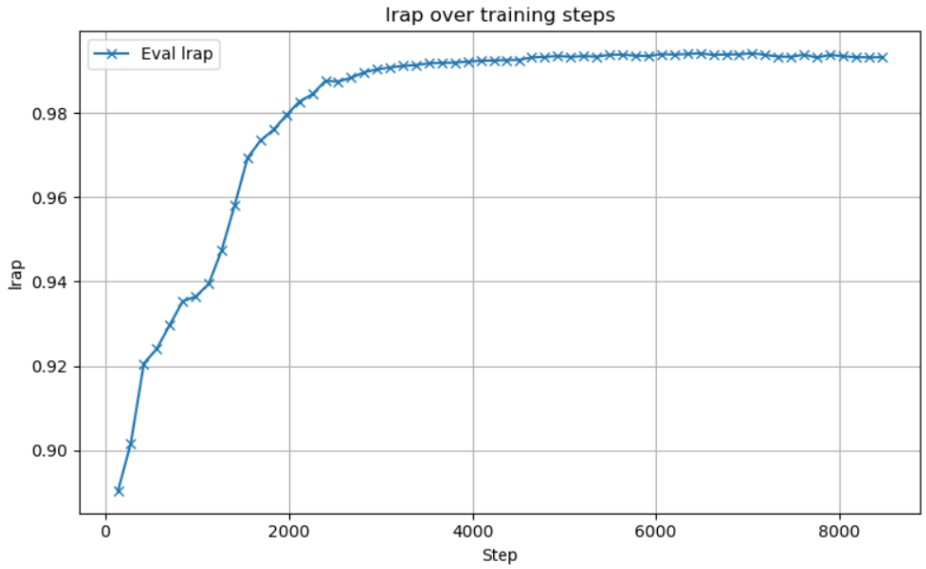

Figure 8: Label Ranking Average Precision (LRAP) progression. The asymptotic approach to 0.99 indicates nearly perfect ranking of true positives above irrelevant predictions.

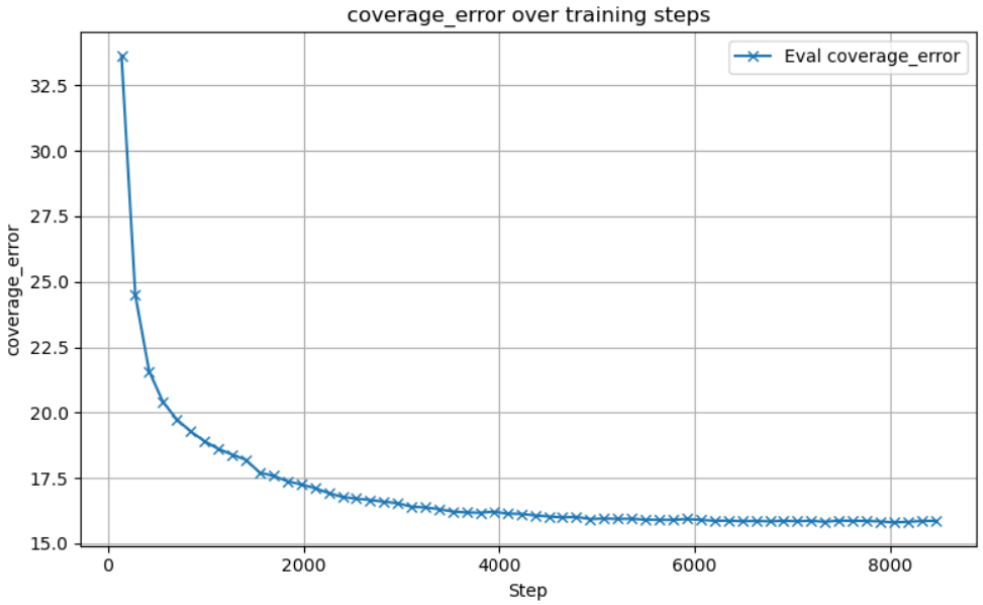

Figure 9: Coverage error reduction. The decrease from 25 to 15 means users need to examine fewer top predictions to catch all errors.

Figures 8 and 9 demonstrate:

- **LRAP** improvement from 0.92 to 0.99 shows excellent ranking capability
- **Coverage Error** reduction by 40% (25→15) enhances practical usability
- Both metrics plateau after 6000 steps, suggesting optimal training duration

7.8 Key Takeaways

The complete results reveal:

- The model achieves 0.98 lenient precision/recall, demonstrating robust detection
- 4% strict-lenient gap confirms handling of noisy labels
- LRAP 0.99 and coverage error 15 indicate production-ready ranking
- All metrics stabilize after ∼6000 steps (∼30 epochs)

8 Conclusion

- The model exhibited strong generalization, with steadily improving validation metrics and no signs of overfitting due to the use of early stopping and low learning rate.
- After applying transfer learning to adapt a pre-trained model to our labeled data, the final model achieved a high strict F1 score (∼0.99), showing strong performance across multiple misconfiguration types.
- Improvements in LRAP and Coverage Error suggest that the model not only predicts correctly but also ranks labels effectively, which is important in multi-label classification tasks.
- The consistent gap reduction between training and validation losses across epochs supports the training's stability and efficiency.

This study shows that a large language model fine-tuned on multi-tool labeled data can effectively identify Kubernetes misconfigurations. By carefully considering the label noise and incompleteness, and applying specialized evaluation metrics, we provide a more realistic assessment of model performance. Our approach highlights the complementary nature of multiple static analysis tools and suggests that AI methods can enhance existing security pipelines.

References

[1] Bridgecrew. Checkov. `https://www.checkov.io`, 2023. Accessed: 2025-07-04.

[2] Michele Chiari, Michele De Pascalis, and Matteo Pradella. Static analysis of infrastructure as code: a survey. *2022 IEEE 19th International Conference on Software Architecture Companion (ICSA-C)*, pages 218–225, 2022.

[3] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

[4] Ehud Malul, Yair Meidan, Dudu Mimran, Yuval Elovici, and Asaf Shabtai. Genkubesec: Llm-based kubernetes misconfiguration detection, localization, reasoning, and remediation. *arXiv preprint arXiv:2405.19954*, 2024.

[5] Akond Rahman, Shazibul Islam Shamim, Dibyendu Brinto Bose, and Rahul Pandita. Security misconfigurations in open source kubernetes manifests: An empirical study. *Proceedings of the ACM on Software Engineering*, 2023.

[6] StackRox. Kubelinter. `https://github.com/stackrox/kube-linter`, 2023. Accessed: 2025-07-04.

[7] Tenable. Terrascan. `https://github.com/tenable/terrascan`, 2023. Accessed: 2025-07-04.