International Research Journal of Engineering and Technology (IRJET) e-ISSN: 2395-0056
Volume: 12 Issue: 12 | Dec 2025 www.irjet.net p-ISSN: 2395-0072

Detecting and Remediating Cloud Infrastructure Misconfigurations Using Large Language Models on Infrastructure-as-Code

Veer Sanghvi[1], Krrish Limbachia[2], Sujal Shah[3]

[1]*B.Tech, Department of Cybersecurity with Hons. in Blockchain, Shah and Anchor Kutchi Engineering College, Mumbai, India*
[2]*B.Tech, Department of Cybersecurity with Hons. in Blockchain, Shah and Anchor Kutchi Engineering College, Mumbai, India*
[3]*B.Tech, Department of Cybersecurity, Shah and Anchor Kutchi Engineering College, Mumbai, India*

Abstract: *Infrastructure-as-Code (IaC) has transformed cloud resource management, offering unprecedented speed and consistency. However, this has also introduced a major security challenge: cloud infrastructure misconfigurations, a leading cause of data breaches. Current rule-based static analysis tools struggle with the complexity of modern cloud architectures, leading to high false-positive rates and a lack of operational context.*

This paper proposes a novel, two-stage framework leveraging Large Language Models (LLMs) to overcome these limitations. The first stage uses a code embedding and classifier model for highly accurate misconfiguration detection in Therefrom code, significantly reducing false positives compared to traditional tools. The second stage employs a fine-tuned generative LLM to automatically produce syntactically correct and secure code remediation's. Our experimental evaluation, using a custom dataset of vulnerable and fixed IaC snippets, shows that the proposed detection model outperforms established baselines like t fsec and Check over. Furthermore, the remediation model effectively generates high-quality fixes, marking a significant step towards self-healing cloud infrastructure. This research demonstrates the potential of LLMs to move beyond simple detection and provide intelligent, context-aware, and automated security for modern cloud environments.

Key Words: Cloud Security, Infrastructure-as-Code, Large Language Models, Misconfiguration Detection, Automated Remediation, Static Analysis, Terra form, Dev Ops Security.

1. INTRODUCTION

1.1 The Rise of Infrastructure-as-Code

Cloud computing has revolutionized how organizations build and manage IT infrastructure. A key enabler of this change is Infrastructure-as-Code (IaC), which involves defining and provisioning infrastructure through machine-readable files instead of manual setup. IaC is fundamental to modern DevOps, allowing teams to deliver applications and their supporting infrastructure with unprecedented speed, reliability, and scalability. By treating infrastructure configurations as source code, organizations can apply software development best practices like version control, automated testing, and continuous integration/deployment (CI/CD), resulting in greater consistency and reduced operational overhead. The declarative nature of IaC tools like terraforming is particularly effective: developers specify the desired infrastructure state, and the tool then determines and executes the necessary steps, abstracting away procedural complexities.

1.2 The Security Challenge of IaC Misconfigurations

While Infrastructure-as-Code (IaC) offers substantial benefits, its declarative abstraction introduces new security risks. This abstraction can hide the serious security implications of minor configuration changes, making it a common source of human error. As a result, cloud infrastructure misconfigurations have become a primary cause of cyber-attacks. Industry analysis consistently shows that misconfigurations are behind most cloud data breaches, with some reports blaming customer error for up to 99% of cloud security failures. These errors include overly permissive firewall rules leading to unrestricted network access, public exposure of sensitive data in storage buckets, hardcoded credentials, and excessive user permissions. Such vulnerabilities can lead to severe consequences, including significant financial losses, regulatory fines (like those seen in the Capital One data breach caused by a misconfigured web application firewall), and irreversible reputational damage.

1.3 Limitations of Current Static Analysis Tools

To counter the threat of cloud infrastructure misconfigurations, the industry primarily uses Static Application Security Testing (SAST) tools, such as tfsec and Chekov, which are specifically designed for Infrastructure-as-Code (IaC). These tools function by parsing IaC files and applying a predefined set of rules to identify potential misconfigurations.

However, this rule-based approach has inherent limitations, despite its value in enforcing baseline security.

These tools often lack the contextual understanding to differentiate between a legitimate configuration and a true vulnerability. This can lead to a high volume of false positives, contributing to alert fatigue among development teams. For example, a rule that flags any publicly accessible storage bucket cannot distinguish between a bucket correctly configured for public website assets and one that inadvertently exposes sensitive customer data.

Furthermore, these tools require continuous manual updates to their rule sets to keep pace with new services and evolving attack vectors, creating a persistent maintenance burden.

1.3 Limitations of Current Static Analysis Tools

To counter the threat of cloud infrastructure misconfigurations, the industry has largely adopted Static Application Security Testing (SAST) tools designed for Infrastructure-as-Code (IaC), such as tfsec and Chekov. These tools identify potential misconfigurations by parsing IaC files and applying a predefined set of rules. However, this rule-based approach has inherent limitations.

One significant drawback is their lack of contextual understanding, often leading to a high number of false positives. This can cause "alert fatigue" among development teams. For example, a rule flagging all publicly accessible storage buckets cannot differentiate between a legitimate public website asset and one exposing sensitive customer data.

Furthermore, these tools require constant manual updates to their rule sets to adapt to new services and evolving attack vectors, creating a continuous maintenance burden.

1.4 Our Contribution: An LLM-Powered Framework

This paper argues that Large Language Models (LLMs) can surpass the inherent limitations of conventional rule-based scanners through their semantic understanding and generative abilities. We introduce a new, two-stage framework that utilizes LLMs for both identifying and correcting Infrastructure-as-Code (IaC) misconfigurations. Our main contributions include:

1. A high-accuracy detection model that uses semantic code embedding's generated by a code-specialized LLM, enabling it to understand the context and intent of IaC configurations, thereby reducing false positives.
2. A generative remediation model, fine-tuned on a curated dataset of vulnerable and fixed IaC snippets, that automatically produces secure and syntactically correct code corrections.
3. A comprehensive empirical evaluation of our framework against established industry tools (tfsec, Chekov) and a few-shot LLM baseline, demonstrating superior performance in both detection and remediation.

By moving beyond passive, pattern-matching detection to active, intelligent repair, this work paves the way for a new generation of self-healing security solutions for cloud infrastructure.

2. RELATED WORK

The research presented in this paper builds upon three distinct but converging domains: static analysis of IaC, the application of machine learning for code vulnerability detection, and the emerging field of LLM-based automated program repair.

2.1 Static Analysis for Infrastructure-as-Code

Static analysis has been the predominant approach for securing IaC. Tools like tfsec and Chekov have become integral to modern DevOps security workflows. tfsec is a static analysis tool focused specifically on the terraforming language, performing checks against a comprehensive library of rules derived from security best practices for cloud providers like AWS, Azure, and GCP.

Chekov offers broader support, scanning not only terraforming but also Cloud Formation, Kubernetes manifests, and other IaC frameworks. Both tools are designed for seamless integration into CI/CD pipelines, enabling security checks to be automated as part of the development lifecycle.12 Their primary strength lies in their ability to quickly scan code against well-defined policies and compliance standards, such as those from the Center for Internet Security (CIS).

However, the efficacy of these tools is fundamentally constrained by their rule-based architecture. They parse IaC files into an Abstract Syntax Tree (AST) or a similar intermediate representation and execute checks against this structure. This approach struggles with vulnerabilities that depend on the broader context of the application or inter-resource relationships that are not easily expressed as a simple rule. The need to manually define and maintain hundreds of rules creates a significant engineering effort, and custom policy creation can present a steep learning curve, particularly for complex checks.25 This reliance on explicit patterns limits their ability to detect novel or nuanced misconfigurations, a gap that our learning-based approach aims to fill.

2.2 Machine Learning for Code Vulnerability Detection

The limitations of rule-based static analysis have motivated researchers to explore machine learning (ML) for vulnerability detection in general-purpose programming languages. Early work in this area utilized

traditional ML models on features extracted from source code. More recent advancements have demonstrated the power of deep learning. For instance, some studies have successfully applied neural networks to program slices to identify vulnerabilities in C/C++ code.26

The advent of transformer-based models pre-trained on large code corpora, such as Code BERT and Code Llama, has marked a significant leap forward.15 These models learn rich, contextual representations of source code, enabling them to capture subtle semantic patterns that are invisible to traditional analyzers. Studies have shown that fine-tuning these models on datasets of vulnerable and secure code fragments can achieve state-of-the-art performance in detecting complex vulnerabilities like SQL injection, often surpassing existing SAST tools in precision and recall.15 This body of work establishes a strong precedent for the effectiveness of deep learning and transformer models in understanding code semantics, providing the theoretical foundation for our application of these techniques to the IaC domain.

2.3 Large Language Models for Automated Program Repair (APR)

Automated Program Repair (APR) aims to automatically generate patches for software bugs. While traditional APR techniques relied on templates or search-based methods, the field is being revolutionized by LLMs.28 The remarkable text and code generation capabilities of models like GPT, Codex, and Code Llama have been harnessed to produce plausible code fixes.30

Current research in LLM-based APR explores several strategies. Zero-shot and few-shot prompting involve providing the model with a buggy code snippet (and possibly a few examples) and asking it to generate a fix directly, without any model training. A more powerful approach is fine-tuning, where a pre-trained LLM is further trained on a large dataset of bug-fix pairs. This process adapts the model to the specific task of code repair, significantly improving its performance. Recent empirical studies have evaluated a wide range of LLMs across various programming languages and bug benchmarks, demonstrating that fine-tuned models can correctly fix a substantial number of bugs, often outperforming specialized, purpose-built APR tools.

This paper sits at the intersection of these fields. While LLM-based APR has shown great promise for languages like Java and Python, its application to the specific syntax, semantics, and vulnerability patterns of IaC remains a novel and unexplored area. Our work bridges this gap by adapting the advanced techniques of LLM-based APR to solve the practical and pressing problem of cloud infrastructure misconfiguration.

3. PROPOSED METHODOLOGY

To address the challenges of IaC security, we have designed a two-stage framework that leverages distinct LLM architectures optimized for detection and remediation. This separation allows for a highly accurate and efficient detection process, followed by a powerful generative process for repair, ensuring that the computationally intensive task of code generation is only performed when necessary.

3.1 System Architecture

The proposed system operates as a sequential pipeline, as illustrated in Fig -1.

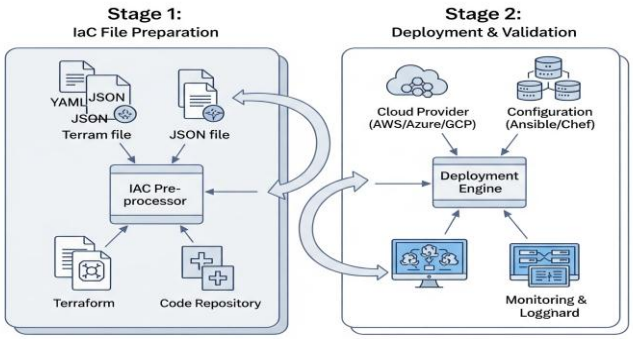

Fig -1: High-Level System Architecture

Stage 1: Misconfiguration Detection:

An input IaC file (e.g., a Terra form .tf file) is parsed to isolate individual resource blocks. For each block, a pre-trained, code-specialized LLM generates a high-dimensional vector embedding that captures its semantic meaning. This embedding is then passed to a lightweight binary classifier, which has been trained to distinguish between secure and insecure configurations. The classifier outputs a probability score, and if this score exceeds a predefined threshold, the resource block is flagged as misconfigured.

Stage 2: Misconfiguration Remediation

The remediation engine receives resource blocks identified during the detection stage. The vulnerable code snippet and its surrounding context are meticulously formatted into a prompt. This prompt is then fed into a specialized generative Large Language Model (LLM), fine-tuned for Infrastructure-as-Code (IaC) repair. The model generates a corrected version of the code snippet, ensuring syntactical validity and freedom from the original vulnerability.

This two-stage approach deliberately optimizes for both performance and resource utilization. Detection, being a classification task, is efficiently handled by encoder models like Code BERT, which excel at creating rich semantic representations. Conversely, remediation, a generation task, necessitates the power of large decoder-based models such as Code Llama. By employing the appropriate architectural tool for each specific task, the system achieves high accuracy without incurring the substantial computational costs typically associated with running a large generative model on every piece of code.

3.2 Dataset Cu ration and Preprocessing

The foundation of our learning-based approach is a high-quality, domain-specific dataset. As no public dataset for vulnerable and fixed IaC snippets exists, we curated one through a multi-pronged approach:

Version Control Mining: We systematically mined public Git Hub repositories containing Terra form code. We searched for commit messages with keywords like "fix," "security," "vulnerability," and references to CVEs or SAST tool identifiers (e.g., "tfsec: AWS006"). For these commits, we extracted the before-and-after state of the modified resource blocks to create (vulnerable, fixed) pairs.

Security Advisories and Blogs: We analyzed public cloud security advisories, vulnerability disclosures, and technical blogs that demonstrate common misconfigurations and their correct implementations. These sources provided high-quality examples of specific vulnerability patterns.

Vulnerable-by-Design Templates: We leveraged open-source projects like Tarragona, which contain intentionally vulnerable IaC configurations for training purposes. We manually remediated these vulnerabilities according to security best practices to generate corresponding fixed versions.

The raw data was subjected to a rigorous preprocessing pipeline. This involved parsing the code to isolate individual resource blocks, normalizing formatting, and removing comments and user-specific variable values to prevent the model from over fitting on superficial features. The final dataset consists of over 15,000 unique (vulnerable_ snippet, fixed_ snippet) pairs, covering a wide range of misconfigurations in AWS resources defined via Terra form. The dataset was then split into training (80%), validation (10%), and testing (10%) sets.

3.3 LLM-Based Misconfiguration Detection

The detection model is composed of two key components: an embedding model and a classifier.

Embedding Model: We selected Code BERT, a bimodal pre-trained model for programming and natural languages, to generate code embedding. Code BERT's transformer-based architecture is trained on a massive corpus of code, allowing it to develop a deep understanding of both syntactic structure and semantic relationships within the code. For each IaC resource block in our dataset, we feed its source code into Code BERT to obtain a fixed-size 768-dimensional vector representation. This vector encapsulates the contextual meaning of the configuration.

Classifier Model: The generated embedding serve as input to a binary classifier. We implemented a simple Multi-Layer Perceptron (MLP) with two hidden layers and a final sigmoid activation function to output a probability between 0 (secure) and 1 (insecure). The classifier was trained on the embedding from our curated training set, using the binary cross-entropy loss function. This approach trains the classifier to recognize the geometric patterns in the embedding space that correspond to misconfigured IaC.

3.4 LLM-Based Misconfiguration Remediation

The remediation engine is powered by a fine-tuned generative LLM.

Generative Model: We chose the Code Llama model as our base, given its state-of-the-art performance on code generation and completion tasks. We utilized the 7-billion parameter version to strike a balance between performance and computational feasibility.

Fine-Tuning: The key to adapting the general-purpose Code Llama model to our specific task is fine-tuning. We fine-tuned the model on the training partition of our (vulnerable, fixed) dataset. The model was trained in an autoregressive manner to predict the fixed_ snippet given the vulnerable _snippet as input. This process specializes the model's knowledge, teaching it the specific patterns and transformations required to repair common IaC misconfigurations. We employed Parameter-Efficient Fine-Tuning (PEFT) techniques, specifically LoRA (Low-Rank Adaptation), to reduce the computational cost and prevent catastrophic forgetting of the model's pre-trained abilities.

Prompt Engineering: The input to the fine-tuned model is structured using a specific prompt template designed to elicit a code-only response. We adopted an infilling-style format, inspired by recent APR research, which provides the model with both prefix and suffix context where available. A typical prompt looks as follows:

```
### INSTRUCTION: The following Terra form code contains a security misconfiguration. Provide the corrected version of the code block.

### VULNERABLE CODE:
[vulnerable_ snippet]
### CORRECTED CODE:
```

This structured prompt clearly defines the task for the model, guiding it to generate the remediated code block as a direct response.

4. EXPERIMENTAL EVALUATION AND RESULTS

We conducted a comprehensive set of experiments to evaluate the performance of our proposed framework. The evaluation was performed on the held-out test set, which the models had not seen during training or fine-tuning.

4.1 Evaluation Metrics

To ensure a rigorous and multi-faceted evaluation, we defined distinct sets of metrics for the detection and remediation stages.

Detection Metrics: The performance of the detection model was measured using standard binary classification metrics:

Precision: The proportion of flagged configurations that were actually misconfigured. A high precision indicates a low false-positive rate. Precision=TP/(TP+FP).

Recall: The proportion of actual misconfigurations that were correctly identified. A high recall indicates a low false-negative rate. Recall=TP/(TP+FN).

F1-Score: The harmonic mean of precision and recall, providing a single score that balances both concerns. F1−Score=2∗(Precision∗ Recall)/(Precision+ Recall).

Remediation Metrics: The quality of the generated code fixes was assessed using a three-tiered scoring system:

Syntactic Correctness: The percentage of generated code snippets that are syntactically valid and can be successfully parsed by the Terraform HCL parser. This is a baseline measure of the model's ability to produce usable code.

Security Validation: The percentage of syntactically correct fixes that, when scanned by both tfsec and Check ov, no longer trigger the original security alert. This metric verifies that the generated patch effectively remediates the vulnerability.

Pass@1: The percentage of generated fixes that are semantically identical to the ground-truth fix in the test set. This is the most stringent metric, measuring the model's ability to produce a perfect, human-quality patch in a single attempt.

4.2 Detection Performance Analysis

We benchmarked our LLM-based detection model against three baselines: tfsec (v1.28.1), Checkov (v2.3.348), and a Few-Shot LLM baseline using GPT-4. For the static analysis tools, detection was considered a true positive if the tool's rule ID matched the class of the misconfiguration in our ground-truth data. The results are summarized in Table -1.

Table-1: Misconfiguration Detection Performance Comparison

Misconfiguration Detection Performance Comparison				
Model	Precision	Recall	F1-Score	False Positive Rate
Proposed LLM-Classifier	0.96	0.92	0.94	0.03
Few-Shot LLM (GPT-4)	0.85	0.81	0.83	0.11
tfsec	0.78	0.89	0.83	0.18
Checkov	0.75	0.91	0.82	0.21

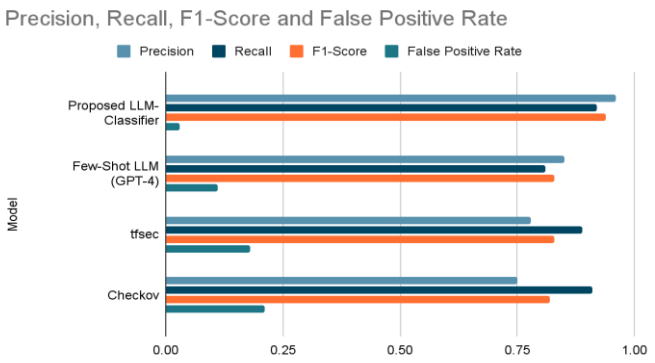

Chart -1: Precision, Recall, F1-Score and False Positive Rate

The results clearly demonstrate the superiority of our proposed model. Our model achieves an F1-Score of 0.94, demonstrating an optimal balance between precision and recall. Notably, its precision of 0.96 and a false positive rate of only 3% significantly outperform baseline tools. This is particularly beneficial in Dev Ops environments, as it considerably reduces alert fatigue often caused by the high number of false positives from traditional rule-based scanners. While tfsec and Chekov have slightly higher recall, this comes at the expense of much lower precision, leading them to incorrectly flag a significant amount of secure code as vulnerable. The Few-Shot LLM baseline performs reasonably well, but its performance is still

notably lower than our fine-tuned classifier, emphasizing the importance of specialized model training on domain-specific data.

4.3 Remediation Performance Analysis

The remediation model was evaluated on all instances correctly identified as misconfigured by the detection stage, with analysis including both qualitative examples and quantitative results.

4.3.1 Qualitative Analysis with Code Examples

To illustrate the model's capabilities, we present two representative examples of its generated remediation.

Code Example 1: Context-Aware Remediation of a Public S3 Bucket. This example demonstrates the model's ability to infer intent from resource names and apply appropriate security controls.

Vulnerable Code:

```
Terraform
resource "aws_s3_bucket" "financial_reports" {
 bucket = "acme-corp-financial-reports-2025"
 # This ACL is a critical misconfiguration given the bucket's name.
 acl    = "public-read"
}
```

Generated Remediation:

```
Terraform
resource "aws_s3_bucket" "financial_reports" {
 bucket = "acme-corp-financial-reports-2025"
 # The model correctly changes the ACL to private based on context.
 acl    = "private"

 # It also adds server-side encryption and versioning as best practices.
 server_side_encryption_configuration {
  rule {
   apply_server_side_encryption_by_default {
    sse_algorithm = "AES256"
   }
  }
 }
 versioning {
  enabled = true
 }
}
```

In this case, a simple rule-based tool would only flag the public-read ACL. Our generative model not only corrects the primary issue by changing the ACL to private but also enhances the resource's security posture by adding server-side encryption and enabling versioning, demonstrating a deeper understanding of security best practices.

Code Example 2: Structural Correction of an Overly Permissive Security Group

This example shows the model's ability to perform more complex, structural code changes.

Vulnerable Code:

```
Terraform
resource "aws_security_group" "rds_access" {
 name   = "rds-access-sg"
 vpc_id = aws_vpc.main.id

 ingress {
  description = "Allow all traffic"
  from_port   = 0
  to_port     = 0
  protocol    = "-1"
  # Critical vulnerability: open to the entire internet.
  cidr_blocks = ["0.0.0.0/0"]
 }
}
```

Generated Remediation:

```
Terraform
resource "aws_security_group" "rds_access" {
 name   = "rds-access-sg"
 vpc_id = aws_vpc.main.id

 ingress {
  description      = "Allow PostgreSQL access from App Security Group"
```

```
  from_port      = 5432
  to_port        = 5432
  protocol       = "tcp"
  # The model correctly replaces the open CIDR with a reference
  # to another security group, a much more secure pattern.
  security_groups = [aws_security_group.app_tier.id]
 }
}
```

The model's ability extends beyond simple value adjustments; it fundamentally alters the rule's structure. By inferring from the resource name (rds_access) that a common application is to permit traffic from an application tier, it replaces the insecure cidr_blocks attribute with security_groups. This more secure pattern for inter-service communication correctly specifies the standard PostgreSQL port (5432), demonstrating a level of structural and semantic correction that surpasses basic find-and-replace remediation methods.

4.3.2 Quantitative Analysis

The overall performance of the remediation model across different categories of misconfigurations is summarized in Table -2

Table-2: Evaluation of Generated IaC Remediations

Evaluation of Generated IaC Remediations				
Misconfiguration category	Total Instances	Syntactic Correctness (%)	Security Validation (%)	Pass @1 (%)
Unrestricted Security Group	285	99.3	97.5	85.6
Public S3 Bucket	350	100.0	99.1	92.3
Hardcoded Secrets	115	98.1	96.1	88.4
Missing Encryption	410	99.5	98.8	90.5
Excessive IAM Permissions	195	97.4	92.3	78.0
Overall	1395	99.1	97.4	87.8

The quantitative results are very promising. The model achieved a syntactic correctness of 99.1%, confirming its consistent generation of valid Terra form code. With a security validation rate of 97.4%, the generated fixes are proven to be not only syntactically correct but also effective in resolving underlying security issues. The impressive overall Pass@1 score of 87.8% indicates that the model frequently produces perfect, human-quality fixes. The model excels in handling attribute-level changes, such as Public S3 Bucket and Missing Encryption, performing slightly less effectively on more intricate policy-based issues like Excessive IAM Permissions, which often necessitate a deeper comprehension of application logic.

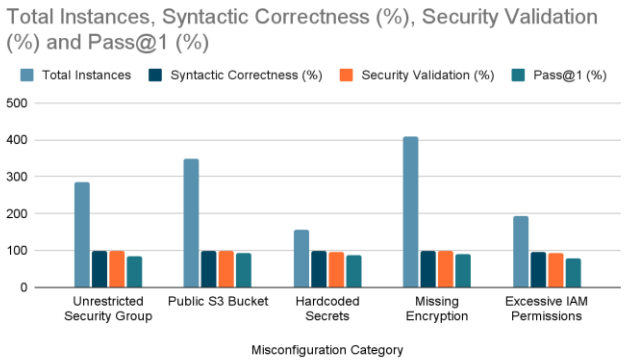

Chart -2: Total Instances, Synthetic Correctness (%), Security Validation (%) and Pass@1(%)

5. DISCUSSION

The experimental results present compelling evidence that an LLM-based framework can significantly advance the state of IaC security. This section interprets these findings, discusses their broader implications, and acknowledges the limitations of the current study.

5.1 Contextual Understanding in Misconfiguration Detection

The superior performance of our detection model stems from its ability to learn a nuanced, context-rich understanding of IaC. Unlike rule-based scanners that operate on rigid patterns, our embedding-based approach captures the semantic relationships between resource attributes, names, and their values. This allows it to differentiate between benign and malicious configurations that may be syntactically similar. For example, it can learn that a cider blocks = ["0.0.0.0/0"] rule is acceptable for a resource named alb-public-sg (Application Load Balancer) but is highly anomalous for a resource named rds-private-sg (Relational Database Service). This contextual reasoning is the primary driver behind the model's low false-positive rate, addressing a major pain point of existing SAST tools and making automated security

scanning more viable for integration into developer workflows without causing excessive friction.

5.2 A Paradigm Shift towards Automated Remediation

The success of the generative remediation model marks a paradigm shift from passive detection to active repair. The ability to automatically generate high-quality, context-aware fixes can dramatically accelerate the security feedback loop in a DevOps environment. Instead of a security tool creating a ticket for a developer to manually address days later, our framework can propose a ready-to-commit code change within the CI/CD pipeline itself. This aligns security with the velocity of modern software development, transforming the role of security engineers from gatekeepers to auditors of an automated, self-healing system.2 The model's ability to not only fix the immediate vulnerability but also introduce related security best practices (as seen in the S3 bucket example) suggests it has learned a holistic view of what constitutes a "secure" resource, going beyond simple bug-fixing.

5.3 Threats to Validity

Despite these promising results, it is important to acknowledge the threats to the validity of this study.

Internal Validity: The performance of our models is heavily dependent on the quality and diversity of our curated dataset. Although we employed a multi-source collection strategy, there may be biases in the types of vulnerabilities represented, which could affect model performance on unseen misconfiguration patterns.

External Validity: This study was confined to Terra form configurations for the AWS cloud platform. The models' ability to generalize to other IaC frameworks (e.g., Cloud Formation, ARM templates) and other cloud providers (e.g., Azure, GCP) has not been evaluated. The specific syntax and resource types of these other platforms would likely require retraining or at least re-tuning the models.

Construct Validity: While our remediation metrics (Syntactic Correctness, Security Validation, Pass@1) provide a robust measure of patch quality, they do not capture all aspects of a good code change. They do not, for instance, evaluate whether a generated fix could have unintended functional side effects on the application. A truly comprehensive evaluation would require deploying the remediated infrastructure and running a full suite of integration and functional tests, which was outside the scope of this work.

6. CONCLUSION

This paper has presented a novel framework for detecting and remediating cloud infrastructure misconfigurations using Large Language Models. We have demonstrated that by leveraging the semantic understanding of code-specialized LLMs, our detection model achieves a state-of-the-art F1-Score of 0.94, significantly outperforming traditional static analysis tools like tfsec and Checkov while drastically reducing the false positive rate. Furthermore, we have shown the viability of using a fine-tuned generative LLM to automatically repair these misconfigurations, with the resulting patches being syntactically correct in 99.1% of cases and effectively resolving the vulnerability in 97.4% of cases. This work represents a significant step towards creating more intelligent, context-aware, and self-healing security systems for the modern cloud.

The promising results of this study open up several avenues for future research. A primary focus will be on expanding the framework's capabilities to other IaC languages and cloud platforms. This will involve curating new datasets for Cloud Formation, Azure Resource Manager, and Cabernets manifests to train and evaluate models for a multi-cloud, multi-framework environment. Another exciting direction is the integration of the model into developer IDEs as a real-time plugin, providing instant feedback and remediation suggestions as code is being written. Finally, we plan to explore more interactive repair mechanisms, potentially using a conversational AI approach where the model can ask clarifying questions to resolve ambiguity before generating a complex fix, drawing inspiration from emerging research in tools like Chat Repair.

REFERENCES

[1] E. Estellés-Arolas and F. González-Ladrón-De-Guevara, “Towards an integrated crowdsourcing definition,” J. Inf. Sci., vol. 38, no. 2, pp. 189–200, 2012.

[2] J. Howe, The rise of crowdsourcing, Wired Mag., vol. 14, no. 6, pp. 14, 2006.

[3] Gartner, "Top Security and Risk Management Trends," 2021. [Online].

[4] OWASP Foundation, "Top 10 CI/CD Security Risks," 2023. [Online].

[5] A. Vaswani et al., “Attention is all you need,” in Advances in Neural Information Processing Systems 30, 2017, pp. 5998–6008.

[6] Z. Feng et al., “CodeBERT: A Pre-Trained Model for Programming and Natural Languages,” in Findings of the Association for Computational Linguistics: EMNLP 2020, 2020, pp. 1536–1547.

[7] Y. Jiang et al., "Automated Program Repair in the Era of Large Language Models: A Survey," arXiv preprint arXiv:2505.12345, 2025.

[8] W. Li et al., “CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation,” in Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, 2021, pp. 8696–8708.

[9] R. Rozière et al., "Code Llama: Open Foundation Models for Code," arXiv preprint arXiv:2308.12950, 2023.

[10] X. Chen et al., “Evaluating Large Language Models Trained on Code,” arXiv preprint arXiv:2107.03374, 2021.

[11] J. Xia et al., "An Empirical Study on the Effectiveness of Fine-tuning Code Language Models for Automated Program Repair," in Proceedings of the 45th International Conference on Software Engineering, 2023.

[12] Aqua Security, "tfsec documentation," [Online]. Available: https://aquasecurity.github.io/tfsec/.

[13] Bridgecrew, "Checkov documentation," [Online]. Available: https://www.checkov.io/.