Evaluating Security Policy Compliance in Infrastructure as Code Generated by Large Language Models

Ryo, Hase; Wang, Ye; Koike-Akino, Toshiaki; Liu, Jing; Parsons, Kieran; Hato, Jumpei

TR2026-036 March 28, 2026

Abstract

Infrastructure as Code (IaC) automates cloud re- source provisioning, yet developing and maintaining IaC scripts remains challenging due to variations in domain-specific lan- guages across providers. Recent advances in large language models (LLMs) offer promise for automating IaC generation, but the security policy compliance of LLM-generated IaC scripts is as important as deployability. In this work, we empirically evaluate configuration-level policy violations in LLM-generated IaC scripts using the IaC-Eval benchmark and Checkov for security policy assessment. Our results indicate that modern LLMs available as of 2025 exhibit improved syntactic correctness and better alignment with user intent, particularly when using retries and error feedback. However, security policy violations persist in generated IaC scripts, typically ranging from around five to fifteen per script across six difficulty levels defined in IaC-Eval. These results underscore the necessity of rigorous verification before deploying generated IaC scripts.

International Symposium on Digital Forensics and Security 2026

Evaluating Security Policy Compliance in Infrastructure as Code Generated by Large Language Models

Ryo Hase
Mitsubishi Electric Corporation
Kamakura, Japan
Hase.Ryo@dc.MitsubishiElectric.co.jp

Ye Wang
Mitsubishi Electric Research Laboratories
Cambridge, MA, USA
yewang@merl.com

Toshiaki Koike-Akino
Mitsubishi Electric Research Laboratories
Cambridge, MA, USA
koike@merl.com

Jing Liu
Mitsubishi Electric Research Laboratories
Cambridge, MA, USA
jiliu@merl.com

Kieran Parsons
Mitsubishi Electric Research Laboratories
Cambridge, MA, USA
parsons@merl.com

Jumpei Hato
Mitsubishi Electric Corporation
Kamakura, Japan
Hato.Jumpei@ea.MitsubishiElectric.co.jp

***Abstract*—Infrastructure as Code (IaC) automates cloud resource provisioning, yet developing and maintaining IaC scripts remains challenging due to variations in domain-specific languages across providers. Recent advances in large language models (LLMs) offer promise for automating IaC generation, but the security policy compliance of LLM-generated IaC scripts is as important as deployability. In this work, we empirically evaluate configuration-level policy violations in LLM-generated IaC scripts using the IaC-Eval benchmark and Checkov for security policy assessment. Our results indicate that modern LLMs available as of 2025 exhibit improved syntactic correctness and better alignment with user intent, particularly when using retries and error feedback. However, security policy violations persist in generated IaC scripts, typically ranging from around five to fifteen per script across six difficulty levels defined in IaC-Eval. These results underscore the necessity of rigorous verification before deploying generated IaC scripts.**

***Index Terms*—Infrastructure as Code, Large Language Model, Static Analysis, Security Policy.**

I. Introduction

A. Background

Cloud computing has been widely adopted as infrastructure for IT systems to realize efficient development and operation. Recent advances in generative AI such as Large Language Models (LLMs) have led cloud providers to offer generative AI services through convenient APIs. Cloud users utilize the services to develop their own solutions; however, they may face challenges when configuring multiple resources composing cloud services. For example, a database service requires resources such as database instances and firewall rules.

Infrastructure as Code (IaC) aims to reduce the burden of manual deployment and management by automating resource provisioning using text files that declaratively specify configurations. For example, Terraform [1] is a provider-agnostic tool that can deploy resources to multiple cloud environments (e.g., Amazon Web Services (AWS) and Microsoft Azure); in this paper, we use the term “IaC script” to refer to such a text file. Although IaC reduces the manual operational workload, it shifts complexity to developing and maintaining IaC scripts, typically written in domain-specific languages (DSLs).

LLMs can assist developers in generating IaC scripts as well as general software code (e.g., Python, TypeScript). However, even when syntactically valid and deployable, generated IaC scripts may include inappropriate resource settings; for example, overly permissive firewall rules. Our research goal is to investigate the effectiveness of using LLMs to generate IaC scripts and to identify the associated risks.

B. Related work

Code generation, including IaC scripts, with LLMs has been widely studied in both academia and industry. Several benchmark datasets have been released to evaluate LLM-based code generation, e.g., HumanEval [2] and SWE-bench [3]. The datasets provide tasks consisting of prompts, test cases, and reference solutions to assess code generation performance. For IaC scripts, IaC-Eval is a dataset designed to evaluate the ability of LLMs to generate IaC scripts [4]. Several LLMs were evaluated with IaC-Eval and found that these models performed worse on IaC-Eval than on HumanEval, which focuses on general coding tasks. However, the models used in the experiments were those available as of 2024 (e.g., GPT-4), and more recent models may exhibit different characteristics.

One research direction for LLM-based IaC script generation is improving deployability by fixing misconfigurations that cause deployment failures. Even semantically correct IaC scripts can fail during deployment because IaC tools cannot capture all constraints in the cloud environment through static analysis. For instance, a validator may not flag an error if a script defines a service for the Japan region while that service is unavailable there. Zhang *et al.* propose IaCGen, which iteratively generates scripts, deploys resources with them, and provides feedback on deployment errors for revisions [5]. Qiu *et al.* present Zodiac, which uses LLMs to discover deployability issue patterns from IaC repositories and validates them by deploying to the real cloud environment [6].

TABLE I
LLMs used for IaC script generation.

Model name	Version	Model name	Version
gpt-5.1-codex-mini	Nov. 2025	devstral-small-2:24b	Dec. 2025
gpt-5-mini	Aug. 2025	mistral-small3.2:24b	Jun. 2025
o4-mini	Apr. 2025	gpt-oss:20b [9]	Oct. 2025
gpt-4.1-mini	Apr. 2025	ministral-3:14b [10]	Dec. 2025
gpt-4o-mini	Aug. 2024	codegemma:7b [11]	Jul. 2024
		codellama:7b [12]	Jul. 2024

While ensuring deployability is essential for producing practically useful IaC scripts, it is also important to identify underlying security risks in LLM-generated IaC scripts. Rahman *et al.* also analyzed security flaws in publicly available IaC scripts and reported issues such as "admin by default," hard-coded secrets, and use of HTTP without TLS [7]. Based on these findings, it is natural to ask whether similar security risks affect IaC scripts generated by LLMs. Although Zhang *et al.* [5] and Qiu *et al.* [6] improve deployability by using pipelines that include real test deployments, such deployments are risky if security flaws remain undetected prior to deployment. Even for testing purposes, security risks in generated IaC scripts may arise in cloud environments, as public cloud providers typically do not offer isolated test environments. Zhang *et al.* analyzed security policy compliance in LLM-generated IaC scripts using their DPIaC-Eval dataset [5]. However, compliance in the earlier IaC-Eval dataset [4] remains unexplored.

C. Contributions

We evaluate the performance and security risks of LLM-generated IaC scripts. Our main contributions are as follows.

- We validate syntax correctness of the configuration in LLM-generated IaC scripts on IaC-Eval dataset, comparing several Azure OpenAI and open models available as of 2025 (e.g., GPT-5.1-codex-mini, devstral-small-2:24b).
- We propose two methods to measure whether generated scripts satisfy user intent: (i) object-level comparison between generated and reference scripts in the IaC-Eval dataset, and (ii) an LLM-based evaluator that scores intent resolution given the prompt and the generated script.
- We employ Checkov [8] to assess whether the IaC scripts comply with security policies; violations detected by Checkov are treated as configuration-level security issues.

II. Experiment design

A. Configurations

We used all 458 prompts from the IaC-Eval dataset to generate IaC scripts with LLMs. Table I lists the models we evaluated. We employed five Azure OpenAI models and six open models accessed via Ollama [13]. Each model generated a single script per prompt. Note that, because LLM outputs are probabilistic, evaluation results may differ if multiple generations per prompt are utilized.

We performed multi-turn generations using the prompt templates provided in the IaC-Eval original paper and its code repository [4]. For the initial generation, we applied their few-shot prompt template, which includes three examples of prompt–output pairs. After each generation, we validated the resulting IaC scripts using OpenTofu as a script validator. If errors were found, we retried generation using their multi-turn prompt template and included the validation error messages as additional input. This retry process was repeated until validation passed or until a maximum of five attempts was reached; scripts were saved as-is after the fifth attempt even if errors remained. For validation, we used OpenTofu [14], an IaC tool compatible with Terraform. Errors were counted by running `tofu init` and `tofu validate` on the generated scripts. Although `tofu init` is a preparatory step before executing `tofu validate`, it can also identify configuration errors.

B. Evaluation metrics

We define four categories of evaluation metrics to assess LLM performance, as described below. Except for configuration validation, metrics are computed from scripts that either passed validation after retries or failed validation even after the maximum number of retry attempts.

1) Configuration validation:
This category measures an ability of LLM to generate syntactically valid scripts. We define the *validation pass rate* as the proportion of generated scripts that pass validation using the commands `tofu init` and `tofu validate` within a specified number of retries. For example, the validation pass rate after the first attempt is 0.50 if 50 out of 100 generated scripts pass validation. If an additional 25 of the remaining scripts pass after one retry, the validation pass rate after the retry (i.e., after the second attempt) is 0.75; in this scenario, 25 scripts still fail even after the retry.

2) Intent resolution:

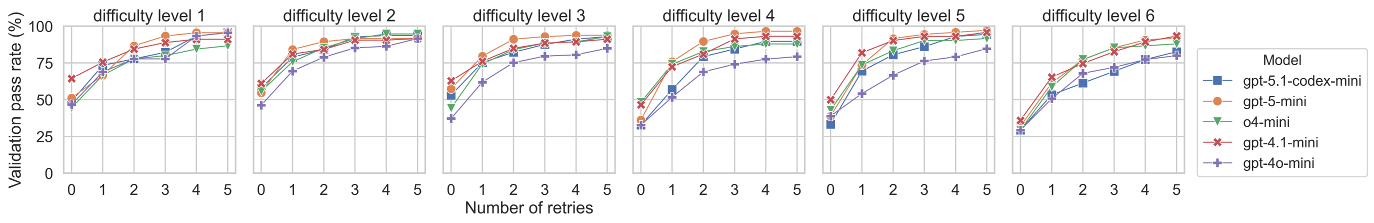
i) Object recall and object precision:
This metric measures how well an LLM includes objects (e.g., providers, resources, data) that appear in the reference scripts of the IaC-Eval dataset. The core idea is that generated IaC scripts better reflect the user's intent when their objects closely match those in the reference scripts. Let $\mathrm{pred}(t)$ denote the count of type t in the generated script, and $\mathrm{gold}(t)$ be the count of t in the reference script. For instance, if a generated script contains two EC2 instances and one VPC, then $\mathrm{pred}(\texttt{aws\_ec2\_instance}) = 2$ and $\mathrm{pred}(\texttt{aws\_vpc}) = 1$. Let $\mathcal{T}$ denote $\{t \mid \mathrm{pred}(t) > 0 \text{ or } \mathrm{gold}(t) > 0\}$. TP (True Positives), object recall, and object precision are obtained as:

$$(\mathrm{TP}) = \sum_{t\in\mathcal{T}} \min\big(\mathrm{pred}(t), \mathrm{gold}(t)\big),$$

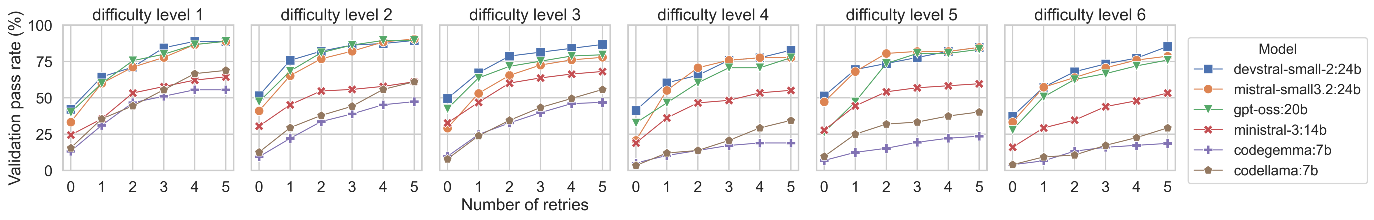
$$(\text{Object recall}) = \frac{\mathrm{TP}}{\sum_{t\in\mathcal{T}} \mathrm{gold}(t)},$$

$$(\text{Object precision}) = \frac{\mathrm{TP}}{\sum_{t\in\mathcal{T}} \mathrm{pred}(t)}.$$

ii) Intent resolution score:
Intent resolution score quantifies how well a generated script aligns with the original user intent in the prompt. We used the



(a) Azure OpenAI models.



(b) Open models.

Fig. 1: Validation pass rate over retry counts.

Intent Resolution Evaluator from the Azure Evaluation SDK [15], with the IaC-Eval prompt and the generated script as input. To mitigate potential bias from using the same model for both generation and evaluation, we selected GPT-5.1 as the evaluation model, which is not included in Table I. The evaluator assigns integer scores from one to five (higher is better). A score of five means the script fully addresses the user intent, while one means it is completely unrelated.

3) Policy compliance:
This category evaluates an LLM's ability to generate scripts containing fewer security policy violations. Checkov detects security policies that are applicable to objects in the scripts and assigns each policy one of three statuses: passed, failed, or skipped. Each policy is also labeled with a severity level: info, low, medium, high, or critical. Note that Checkov cannot evaluate scripts with parsing errors; such scripts were omitted from our calculations. We used the built-in policies that Checkov applies by default. We computed the following metrics from Checkov evaluations of generated scripts for each LLM:

- *Total number of failed policies*: the total count of failed policy instances across all generated scripts.
- *Number of failed policy types*: the count of unique policy types that failed across all scripts.
- *Severity distribution of failed policies*: the proportion of failed policies by severity level.

III. Results

The experimental results are presented below. IaC-Eval treats prompts as specifications for generating IaC scripts, and prompts are classified into six difficulty levels based on their complexity (e.g., number of resources and lines of code). We report the results broken down by difficulty level because we found the difficulty level affects performance on some metrics.

A. Configuration validation

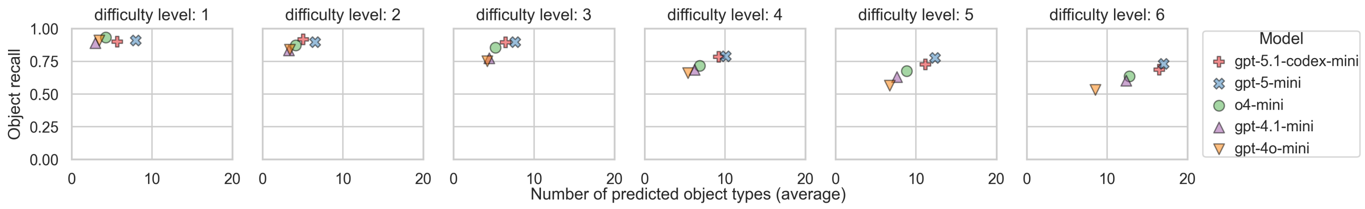
Figures 1 (a) and (b) show validation pass rates across retry counts for Azure OpenAI models and open models, respectively. For both model groups, applying retries with validation feedback improved the validation pass rate; the initial generation, however, performed relatively worse. Overall, difficulty level affected performance on the initial template generation. After several retry attempts, the effect of difficulty level on the validation pass rate diminished. Nevertheless, higher-difficulty tasks required more retries to reach comparable pass rates.

With sufficient retries, some recent models achieved high pass rates of 80–90% even on more complex tasks. A large proportion of the IaC scripts generated by those models were syntactically correct and therefore suitable for analysis by Checkov, which only evaluates syntactically valid IaC scripts. Considering the trends in Figures 1 (a) and (b), increasing the number of retries beyond five may improve the validation pass rate while incurring higher computational costs.

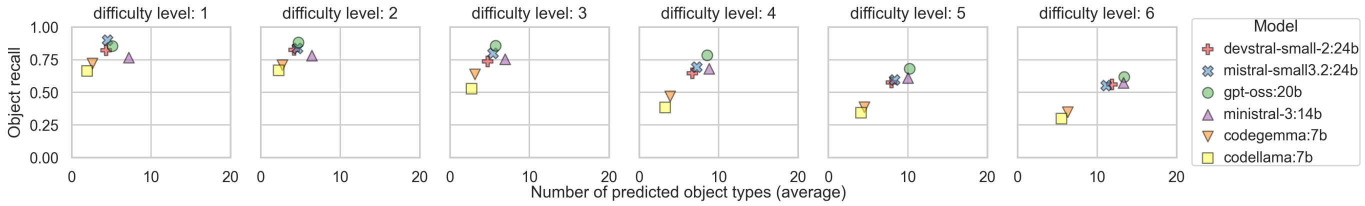
B. Intent resolution

i) Object recall and precision
Figure 2 (a) and (b) plot object recall against the average number of predicted object types for Azure OpenAI models and open models, respectively. These plots indicate that models predicting a larger number of resources tend to achieve higher object recall, suggesting they cover many of the objects defined in the reference templates of the IaC-Eval dataset. However, recall generally decreases as difficulty level increases.

Figure 3 (a) and (b) plot object precision against the average number of predicted object types for Azure OpenAI models and open models, respectively. The precision trends are the inverse of the recall trends. Some relatively older models (e.g., GPT-4o-mini and codellama-7b) outperformed more recent models (e.g., GPT-5.1-codex-mini and devstral-small-2). One possible explanation is that newer models predict a larger set of object types, including items absent from the reference scripts, which reduces precision. The results suggest that balancing object precision and object recall is difficult because the number of objects predicted varies across LLMs.

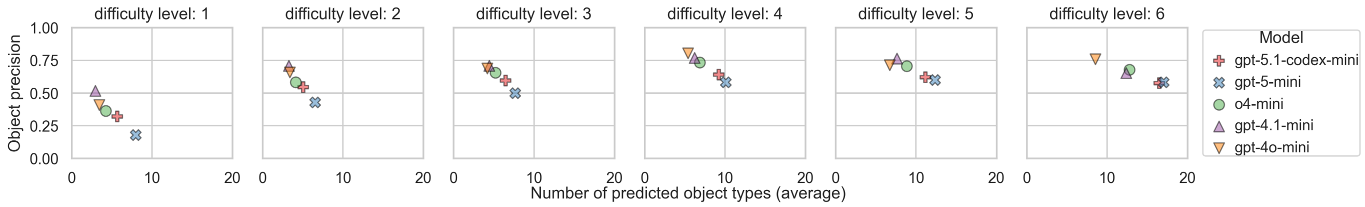


(a) Azure OpenAI models.

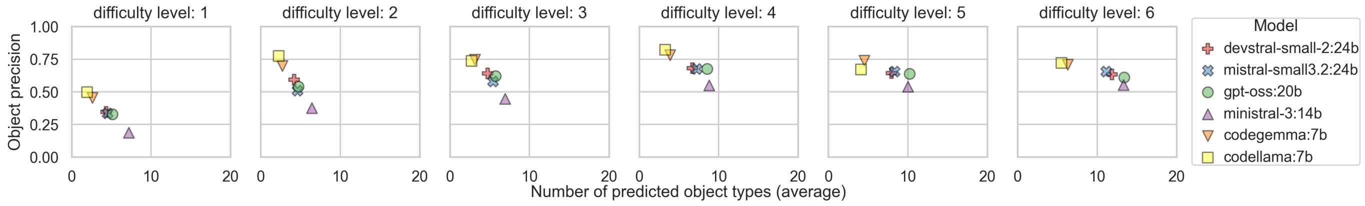


(b) Open models.

Fig. 2: Object recall over average number of predicted object types.



(a) Azure OpenAI models.



(b) Open models.

Fig. 3: Object precision over average number of predicted object types.

ii) Intent resolution score

Figure 4 (a) and (b) show intent resolution scores produced by the evaluator for Azure OpenAI models and open models, respectively. Figure 4 (a) indicates that most templates generated by relatively recent Azure OpenAI models (e.g., gpt-5.1-codex-mini and gpt-5-mini) received a score of five, suggesting strong alignment with the intent expressed in the original user prompts. The adverse effect of increasing difficulty was limited for these recent models, whereas scores for older models (e.g., gpt-4o-mini) declined as difficulty increased.

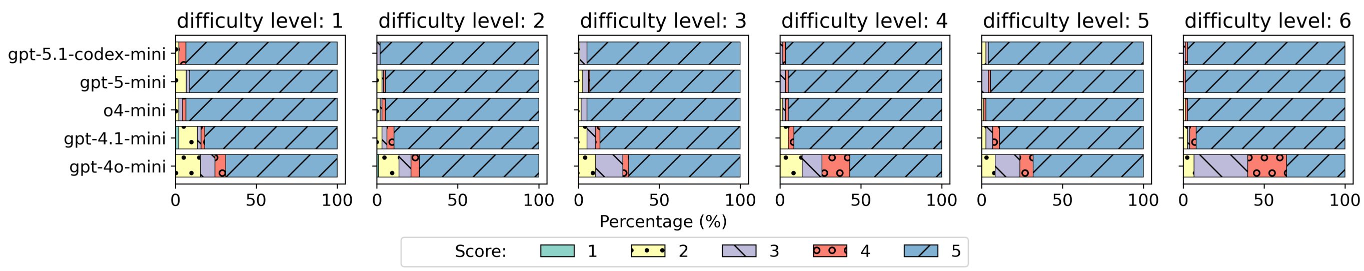
Figure 4 (b) shows that larger open models (e.g., devstral-small-2:24b, ministral-small3.2:24b, gpt-oss:20b) outperformed smaller ones. Difficulty level tended to lower scores for nearly all open models. gpt-oss:20b (released by OpenAI) achieved the highest score among the open models.

Given that gpt-oss:20b obtained the highest score among open models, using GPT-5.1 (an OpenAI model) as the evaluator may have biased results toward OpenAI outputs. Future work should incorporate evaluators from other providers and include human evaluations to determine whether such bias is present. Moreover, the set of IaC generation models should be expanded, particularly to include models released by cloud providers other than Azure OpenAI. Such expansion would increase diversity and help reveal how intent resolution and evaluation outcomes vary across different models.

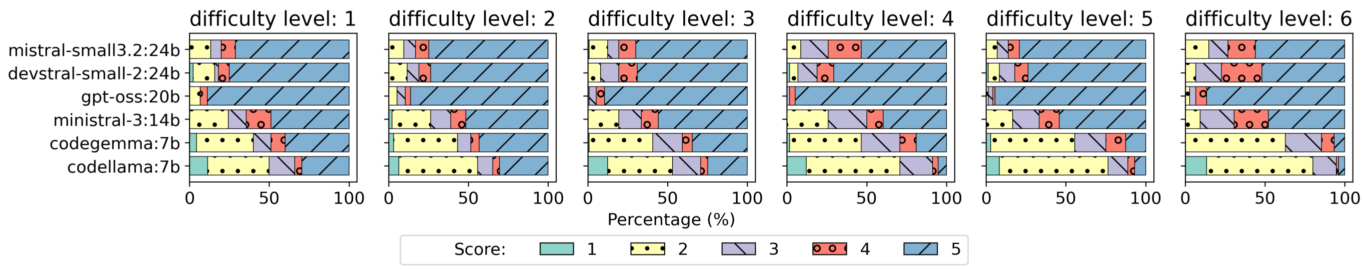
C. *Policy compliance*

Table II and III present the number of failed Checkov policies by severity for Azure OpenAI models and open models, respectively. "total" denotes total number of failed policies, and "type" denotes the number of failed policy types. No failures were observed at the "critical" severity, so that label is omitted from the tables.

Figure 5 (a) and (b) show the severity distribution of failed policies for Azure OpenAI models and open models, respectively. The distributions are similar across models: roughly



(a) Azure OpenAI models.



(b) Open models.

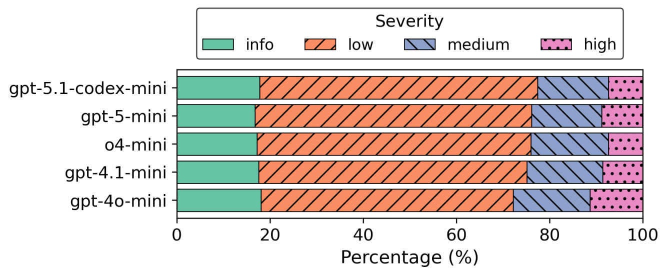
Fig. 4: Scores evaluated by intent resolution evaluator.

TABLE II
NUMBER OF FAILED POLICIES PER SEVERITY (AZURE OPENAI MODELS).

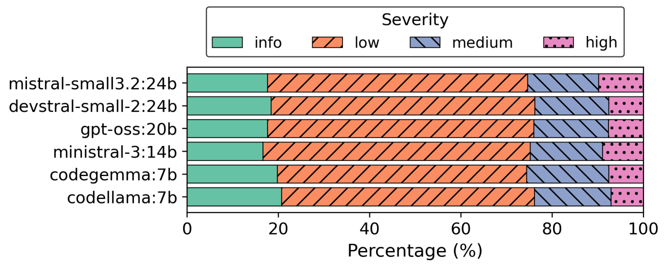
Model	info	low	med.	high	total	types
gpt-5.1-codex-mini	613	2,050	524	253	3,440	198
gpt-5-mini	661	2,329	592	343	3,925	191
o4-mini	523	1,790	504	226	3,043	184
gpt-4.1-mini	528	1,727	490	258	3,003	188
gpt-4o-mini	552	1,653	503	348	3,056	172

TABLE III
NUMBER OF FAILED POLICIES PER SEVERITY (OPEN MODELS).

Model	info	low	med.	high	total	types
devstral-small-2	587	1,840	516	243	3,186	197
mistral-small3.2	572	1,852	507	316	3,247	198
gpt-oss	561	1,864	520	245	3,190	196
ministral-3	544	1,923	518	297	3,282	194
codegemma	476	1,312	432	182	2,402	185
codellama	460	1,233	375	159	2,227	189



(a) Azure OpenAI models.



(b) Open models.

Fig. 5: Ratio of severity assigned to failed Checkov policies.

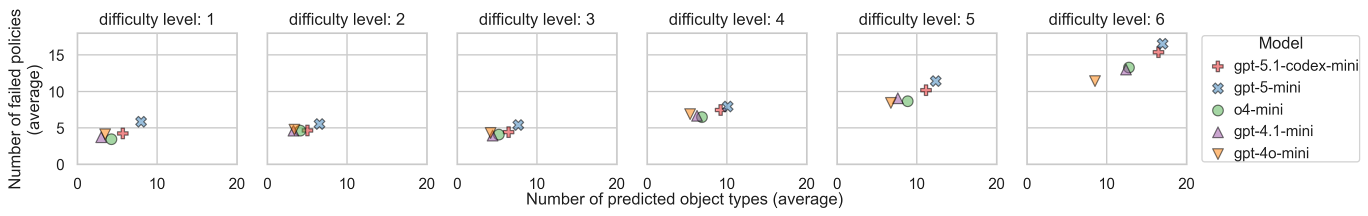
70% of failed policies were classified as info or low, while about 30% were classified as medium or high. Given these similar proportions, models that predict more resources tend to produce a larger absolute number of risky policies and thus warrant greater attention from developers.

Figure 6 (a) and (b) plot the number of failed policies against the number of predicted object types for Azure OpenAI models and open models, respectively. Models that predicted more objects tended to produce more failed policies. Notably, many failed policies were observed even for relatively recent models such as GPT-5.1-codex-mini and GPT-5-mini. These results suggest that developers should inspect which policies fail for templates generated by models regardless of the release dates of the models.

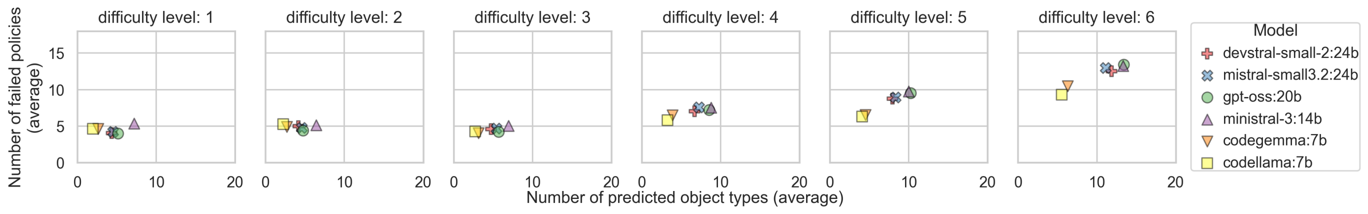
There are several limitations associated with these findings. For example, we excluded IaC scripts that failed to parse because Checkov does not report policy violations for unparsable inputs; however, those excluded scripts may still contain violations. Moreover, our analysis is limited to the default policy set provided with Checkov. Using additional security policy checkers (e.g., Trivy [16] or KICS [17]) could help identify further risks in the generated IaC scripts.

IV. CONCLUSION AND FUTURE WORK

This paper presented evaluation results on the performance and security risks of LLM-generated IaC scripts. The results show that, given sufficient retries and detailed error feedback,



(a) Azure OpenAI models.



(b) Open models.

Fig. 6: Average number of failed Checkov policies over the average number of predicted object types.

many recent models can produce syntactically correct IaC scripts that align with the user intent across difficulty levels in the IaC-Eval dataset. This is a promising outcome, as a large proportion of generated IaC scripts are syntactically valid, enabling the application of security policy checkers such as Checkov to detect potential violations. Our experiments show that Checkov identifies numerous policy violations even in outputs from powerful, recently released models. Therefore, we recommend that developers always verify and resolve any policy violations in LLM-generated IaC scripts before deployment, even for testing purposes.

Potential directions for future work include the following. A detailed analysis of the failed policies detected in this study is required to clarify the actual risks they flag, and IaC scripts with parsing errors should also be examined to identify additional risks. The multi-turn strategy used to improve validation pass rates could be adapted to align generated IaC scripts with detected policies, but its practical effectiveness should be evaluated since retries may introduce new validation failures or policy violations. Furthermore, ensuring that generated scripts comply with detected policies without compromising the original specifications and user intent remains challenging but essential. Additional models, particularly those released by different providers, should be used for both generation and evaluation to increase diversity and clarify the effects of potential bias when models serve as evaluators. Employing various security policy checkers and using datasets that target different IaC languages could further enhance understanding and support improved policy compliance in IaC scripts.

References

[1] HashiCorp, "Terraform," https://developer.hashicorp.com/terraform (Accessed: February 13, 2026).

[2] M. Chen *et al.*, "Evaluating large language models trained on code," *arXiv preprint, arXiv:2107.03374*, 2021.

[3] C. E. Jimenez *et al.*, "SWE-bench: Can language models resolve real-world github issues?" in *The Twelfth International Conference on Learning Representations*, 2024.

[4] P. T. Kon *et al.*, "Iac-eval: A code generation benchmark for cloud infrastructure-as-code programs," *Advances in Neural Information Processing Systems*, vol. 37, pp. 134 488–134 506, 2024.

[5] T. Zhang, S. Pan, Z. Zhang, Z. Xing, and X. Sun, "Deployability-centric infrastructure-as-code generation: An llm-based iterative framework," *arXiv preprint, arXiv:2506.05623*, 2025.

[6] Y. Qiu, P. T. J. Kon, R. Beckett, and A. Chen, "Unearthing semantic checks for cloud infrastructure-as-code programs," in *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, 2024, pp. 574–589.

[7] A. Rahman, C. Parnin, and L. Williams, "The seven sins: Security smells in infrastructure as code scripts," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, May 2019, pp. 164–175.

[8] PrismaCloud, "Checkov," https://www.checkov.io/ (Accessed: February 13, 2026).

[9] S. Agarwal *et al.*, "gpt-oss-120b & gpt-oss-20b model card," *arXiv preprint, arXiv:2508.10925*, 2025.

[10] A. H. Liu *et al.*, "Ministral 3," *arXiv preprint, arXiv:2601.08584*, 2026.

[11] C. Team *et al.*, "Codegemma: Open code models based on gemma," *arXiv preprint, arXiv:2406.11409*, 2024.

[12] B. Rozière *et al.*, "Code llama: Open foundation models for code," *arXiv preprint, arXiv:2308.12950*, 2024.

[13] "Ollama," https://ollama.com/ (Accessed: February 13, 2026).

[14] "Opentofu," https://opentofu.org/ (Accessed: February 13, 2026).

[15] Microsoft, "Agent evaluators," https://learn.microsoft.com/en-us/azure/ai-foundry/concepts/evaluation-evaluators/agent-evaluators?view=foundry-classic (Accessed: February 13, 2026).

[16] Aqua Security, "Trivy," https://trivy.dev/ (Accessed: February 13, 2026).

[17] Checkmarx, "Kics," https://kics.io/ (Accessed: February 13, 2026).