AIaCGateGuard: A Security-First Pipeline for Benchmarking LLM- and SLM-Generated Infrastructure-as-Code

Francis Luis Santos Vargas[1], **Rodrigo Brandão Mansilha**[1], **Diego Kreutz**[1]

[1]AI Horizon Labs, PPGES, Universidade Federal do Pampa (UNIPAMPA)

{francisvargas.aluno,rodrigomansilha,diegokreutz}@unipampa.edu.br

Abstract. *We present* AIaCGateGuard*, an open-source, fully automated pipeline for evaluating the security compliance of Infrastructure-as-Code (IaC) generated by Large and Small Language Models (LLMs/SLMs). Unlike existing IaC generation benchmarks, which assess only syntactic validity or functional correctness,* AIaCGateGuard *integrates dual-tool static security analysis (Checkov and Trivy) directly into a GitLab CI/CD pipeline, producing per-model, per-scenario security compliance reports. The tool supports plug-and-play integration of new LLMs via REST API and new SLMs via local inference (`llama-server`), configurable prompt strategies and security specificity levels, optional infrastructure validation and deployment stages (plan-real, apply, destroy), and a Model Context Protocol (MCP) server for conversational interaction with the pipeline. We demonstrate the tool on seven state-of-the-art models across 17 AWS Terraform scenarios, producing 3,570 automated evaluations and exposing security compliance gaps invisible to syntactic-only validation.* AIaCGateGuard *is publicly available, including pipeline code, prompt templates, scenario definitions, and a video demonstration of installation and usage.*

1. Introduction

Automated generation of Infrastructure-as-Code (IaC) by Large Language Models (LLMs) and Small Language Models (SLMs) has become common practice in DevOps pipelines, accelerating cloud resource provisioning [Morris 2020]. However, evaluating the *security* of generated code remains largely a manual process: developers must run security scanners (Checkov, Trivy, tfsec) individually over each generated configuration, compare results across different models and prompts, and repeat the process for every new model version or scenario, an effort that does not scale for systematic evaluations involving multiple models, multiple prompt strategies, and multiple security specificity levels.

Existing IaC generation evaluation tools, such as IaC-Eval [Kon et al. 2024] and DPIaC-Eval [Zhang et al. 2025], focus on syntactic validity or deployability, without integrating multi-tool security analysis as a primary evaluation criterion. As a result, security teams seeking to assess whether a given LLM or SLM is secure enough to generate IaC in production lack a ready-to-use tool that automates this process end-to-end.

We present AIaCGateGuard, an open-source pipeline that addresses this gap by automating generation, syntactic validation, multi-tool security analysis (Checkov and Trivy), and scanner-feedback revision for Terraform configurations generated by LLMs

and SLMs. The tool is implemented as a matrix of parallel GitLab CI/CD jobs, allowing researchers and DevSecOps teams to evaluate new models, new scenarios, or new prompt strategies without modifying pipeline code, declarative configuration is sufficient.

The main contributions of AIACGATEGUARD as a tool are:

(i) a fully automated six-stage GitLab CI/CD pipeline (Generate, Validate, Scan, Evaluate, Report, Revise) with native LLM (REST API) and SLM (`llama-server`) support;
(ii) dual-scanner integration (Checkov for policy compliance, Trivy for vulnerability severity) with automatic metric aggregation; Wilson CIs and Bonferroni chi-square testing are available as post-processing scripts on the same `summary.json` artifacts;
(iii) configurable prompt specificity (L1–L3) and zero/few-shot strategies, enabling systematic experimentation without code changes;
(iv) optional scanner-feedback revision stage that re-prompts models with raw Checkov/Trivy findings for self-correction;
(v) optional infrastructure stages (*plan-real*, *apply*, *destroy*) extending evaluation to real AWS provisioning, all manually triggered with no code changes required;
(vi) a Model Context Protocol (MCP) server exposing the pipeline as a conversational interface, allowing any MCP-compatible client to trigger generation, evaluation, and optional deployment via natural-language instructions.

The tool was validated in a case study involving seven models, including three LLMs and four SLMs, across 17 AWS Terraform scenarios, resulting in 3,570 automated evaluations [Vargas et al. 2026b]. The results, presented in Section 4, show that syntactic validity and security compliance are largely independent properties in LLM-generated IaC. A preliminary version was also evaluated in [Vargas et al. 2026a] using four S3 scenarios and six models, further supporting the tool's applicability.

The complete source code, prompt templates, scenario definitions, and installation and usage video are publicly available.[1] The remainder of the paper presents the conceptual solution, functional prototype, evaluation and demonstration, followed by the final considerations.

2. Conceptual Solution

AIACGATEGUARD is implemented as a six-stage pipeline running as a parallel job matrix in GitLab CI/CD, illustrated in Figure 1. Each stage is a self-contained, independently testable component, communicating via structured JSON artifacts persisted between pipeline jobs.

2.1. Optional Stage 0: LLM-Consensus Prompt and Scenario Design

Stage 0 is an optional pre-processing utility that generates and validates scenario definitions and prompt templates via multi-LLM consensus. Given a task description and a target AWS resource family, it queries a configurable panel of LLM evaluators (default: six providers) to (i) draft system prompts at each specificity level (L1–L3) and (ii) assess

[1] https://gitlab.com/security-iac/security-first-evaluation-of-text-to-terraform/-/tree/feat/sf-sbseg

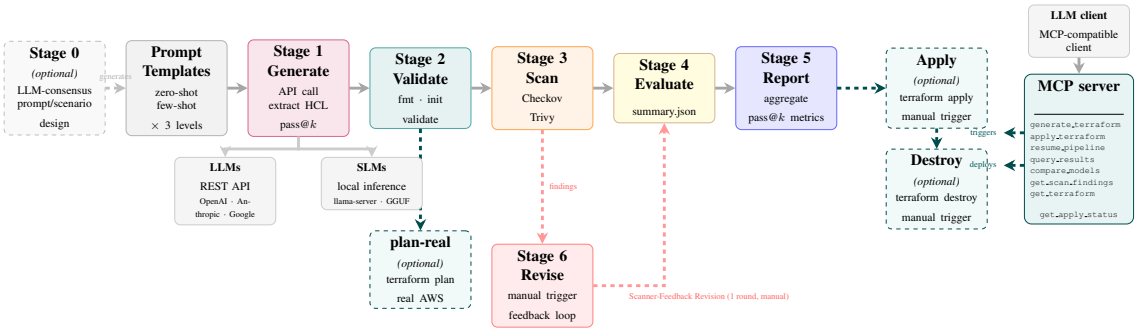


Figure 1. AIACGATEGUARD pipeline architecture. Stage 0 (dashed, optional) generates scenarios and prompts via multi-LLM consensus. Stages 1–5 run automatically as a parallel GitLab CI/CD job matrix. Stage *plan-real* (dashed, optional) executes `terraform plan` against a real AWS account after Stage 2, detecting provider-level errors invisible to syntactic validation. Stage 6 is triggered manually by design, keeping a human reviewing scanner findings before any re-prompt, and performs a single re-prompting round per invocation; it does not loop automatically, and the user may trigger it again to request an additional round (Section 5 discusses automated multi-round revision as future work). Stages *Apply* and *Destroy* (dashed, manual) deploy and tear down approved artifacts in a sandbox AWS environment. The MCP server (right) exposes the pipeline as a conversational interface: `generate_terraform` triggers Stages 1–4 via the GitLab API and returns the approved artifact; `apply_terraform` deploys it locally to the user's AWS environment.

whether a scenario exercises security-relevant Terraform constructs. A candidate is accepted only upon unanimous agreement; disagreement flags it for manual review. Stage 0 is fully decoupled from the core pipeline, its outputs are the same `scenarios.yaml` and prompt template files consumed by Stage 1, so manually authored scenarios can be used without it.

2.2. Pipeline Stages

Table 1 summarises all pipeline stages. Stages 1–5 run automatically as a parallel job matrix; the remaining stages are optional and manually triggered. Each stage communicates via structured JSON artifacts persisted between jobs.

2.3. Extensibility

AIACGATEGUARD was designed for extension without modifying pipeline code:

- **New models** are added via a YAML configuration entry specifying provider type (REST API or local inference), endpoint, and authentication; no code change is required for models implementing the standard chat-completion interface.
- **New scenarios** are added as declarative task definitions (task description, target resource family, expected Checkov/Trivy identifiers), independent of the prompt template engine, and may optionally be drafted via Stage 0.
- **New scanners** are added by implementing a normaliser adapter that maps tool-specific output to the internal pass/fail/severity schema consumed by Stage 4.
- **New prompt strategies and security levels** are added as template files, parameterised by the same scenario definitions used by existing levels (L1–L3), and may optionally be drafted via Stage 0.

Table 1. AIaCGateGuard pipeline stages († optional/manual).

Stage	Function	Artifact
1 — Generate	LLM/SLM call (REST or local); HCL extraction; pass@k	`main.tf`
2 — Validate	`fmt/init/validate` in pinned Docker container	`validate.json`
plan-real†	`terraform plan` vs. real AWS; detects provider-level errors	`plan_real.json`
3 — Scan	Checkov [Bridgecrew 2024] + Trivy [Aqua Security 2024]; per-check pass/fail	`scan.json`
4 — Evaluate	Aggregates stages 2–3 (+ plan-real) into per-generation record	`summary.json`
5 — Report	pass@k rates, compliance heatmap, cross-model comparison tables	`reports/`
6 — Revise†	Re-prompts model with raw scanner findings; artifacts re-enter Stage 2	updated artifacts
Apply†	`terraform apply` for approved artifacts (sandbox only)	`apply.json`
Destroy†	`terraform destroy`; mandatory cleanup; failure blocks pipeline	`apply.json`

3. Functional Prototype

Figure 2 presents the deployment architecture of AIACGATEGUARD using the C4 Container model [Brown 2018], showing the distribution of components across three zones: *Local* (developer machine, MCP server, optional local Terraform), *GitLab CI/CD* (pipeline, registry, Pages), and *AWS Cloud* (S3 state backend and sandbox resources).

3.1. Deployment Requirements

The experiments reported in the companion paper [Vargas et al. 2026b] used a single AWS EC2 `c8g.16xlarge` instance (Graviton4, 64 vCPUs, 128 GB RAM) to host the GitLab CI/CD runner and up to four concurrent `llama-server` processes for local SLM inference; this reflects the infrastructure used in our evaluation, and is not a minimum requirement for running the pipeline. The number of parallel CI/CD jobs and of concurrent SLM processes is configurable, via the GitLab job matrix and `models.yaml` respectively, so the pipeline can run on more constrained hardware by reducing parallelism; this configuration is described in the project documentation. LLM-only deployments require only a GitLab runner with internet access to provider APIs. All components run inside Docker containers with pinned tool versions (Terraform 1.7.5, AWS Provider 5.100.0, Checkov $\geq$3.2.0, Trivy 0.69.3), ensuring reproducible execution.

3.2. MCP Interface

AIACGATEGUARD includes a Model Context Protocol (MCP) [Anthropic 2024] server that exposes the pipeline as a conversational interface, allowing any MCP-compatible client to interact with the tool through natural-language instructions. The server implements the stdio transport and registers two categories of interface elements.

Resources expose read-only views of evaluation artifacts: `benchmark://scenarios` lists configured evaluation scenarios; `benchmark://models` lists models with available artifacts; and `benchmark://results/{model_id}` returns all

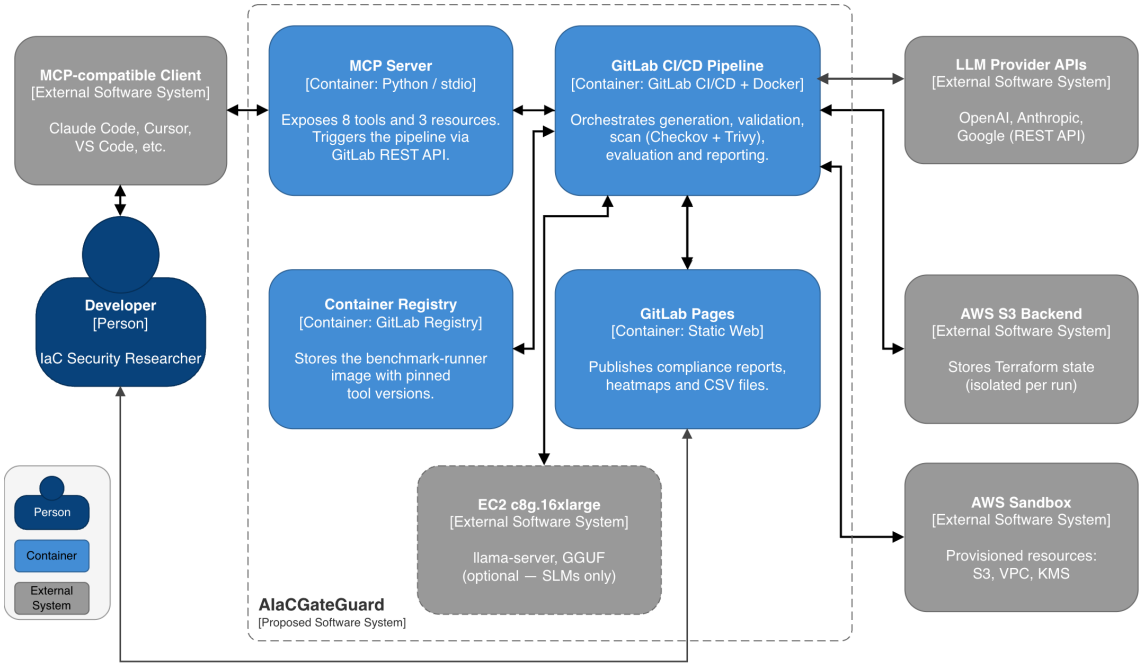


Figure 2. Global deployment architecture of AIACGATEGUARD (C4 Container model). The *Local* zone shows components running on the developer's machine: an MCP-compatible client communicates with the MCP server, which triggers the GitLab CI/CD pipeline and can invoke `apply_terraform` locally. The *GitLab CI/CD* zone hosts the benchmark pipeline, container registry, and GitLab Pages for report publication. The *AWS Cloud* zone shows the sandbox account used by the optional infrastructure stages. LLM provider APIs and an optional EC2 instance for SLM inference are external dependencies.

`summary.json` records for a given model, enabling aggregate queries without direct filesystem access.

Tools expose active operations:

- `generate_terraform` accepts a natural-language infrastructure requirement and an optional security level (`basic` or `detailed`), triggers Stages 1–4 via the GitLab API, polls for completion, and returns the approved `main.tf` with per-scanner compliance status. If the generated code fails any scanner, structured findings are returned so the user can reformulate the requirement.
- `resume_pipeline` resumes polling for a previously triggered pipeline by ID, avoiding redundant pipeline dispatches when the initial polling window expires.
- `apply_terraform` accepts the approved Terraform code and AWS credentials (collected conversationally by the LLM client), executes `terraform init`, `plan`, and `apply` locally, and reports per-step output and provisioning status.
- Read-only tools (`query_results`, `compare_models`, `get_scan_findings`, `get_apply_status`, `get_terraform`) support interactive exploration of benchmark results.

The MCP server acts solely as a protocol adapter between the LLM client and the GitLab API: all generation and scanning work is delegated to the existing pipeline stages, preserving the security guarantees of pinned tool versions and isolated Docker containers regardless of how the pipeline is invoked.

3.3. Main Functionalities

Configuration and Execution. A complete evaluation run is configured through three declarative files: `models.yaml` (model list, provider, authentication), `scenarios.yaml` (target resource family, Checkov/Trivy identifiers per level), and `benchmark.yaml` (global parameters: k, prompt strategies, security levels, revision threshold). Execution is triggered by pushing to the configured GitLab branch, which launches the Stage 1–5 job matrix automatically. For local testing, the pipeline runs via the `benchmark-runner` Docker image, which bundles Terraform, Checkov, and Trivy.

Live Monitoring. Each pipeline job streams structured logs to the GitLab CI/CD console, reporting per-generation status in real time. Users can monitor progress per model via the GitLab pipeline graph without inspecting individual job logs.

Output Artifacts. Stage 5 (Report) automatically produces, with no manual step, three artifacts directly from the Stage 4 `summary.json` records: (i) per-model summary tables (CSV and Markdown) reporting validate%, Checkov%, and Trivy% rates; (ii) a per-scenario compliance heatmap highlighting where failures concentrate; and (iii) raw `summary.json` artifacts per generation, enabling custom downstream analysis. Wilson confidence intervals and Bonferroni chi-square significance testing are deliberately kept outside this automatic stage and run as a separate, manually executed post-processing script over the same `summary.json` artifacts (see Section 5 for the rationale and future integration plans).

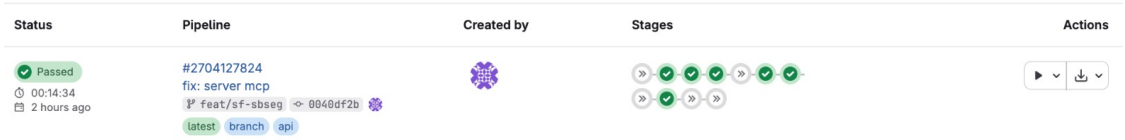
Stage 6 is triggered manually by design, so a human reviews the Checkov/Trivy findings before any re-prompt is sent to the model, rather than letting the pipeline re-prompt unattended. When triggered, the tool identifies validate-passing generations with scanner findings, embeds the raw Checkov/Trivy output in a revision prompt, and re-queries the model. Each invocation performs a single revision round; the user re-triggers the stage to request an additional round. Stage 5 produces a side-by-side Round 1-vs-Revision comparison table. The tool supports three use cases: model selection, prompt engineering evaluation (L1→L3 gains), and regression testing across model versions.

Conversational IaC Generation via MCP. When the MCP server is active, users interact through natural-language instructions in any MCP-compatible client. In the *generation phase*, the user describes the desired infrastructure and the client invokes `generate_terraform`, which triggers Stages 1–4 and returns the `main.tf` with per-scanner compliance status, or structured findings if a scanner fails, so the user can refine the requirement. In the optional *deployment phase*, the client collects AWS credentials conversationally and invokes `apply_terraform`, reporting `terraform init`, `plan`, and `apply` output.

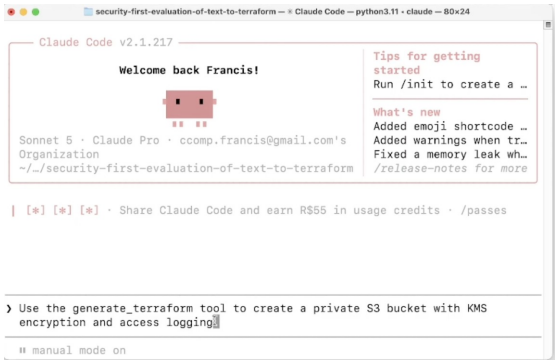
4. Evaluation

Here we demonstrate the utility of AIACGATEGUARD through a live scenario using `benchmark_test.yaml` (3 scenarios, 1 LLM, $k = 1$), executable on any GitLab shared runner without specialised hardware; a full pipeline run (Stages 1–5) completes in approximately 10–15 minutes. A video demonstrating installation, configuration, and a complete pipeline execution is publicly available.[2]

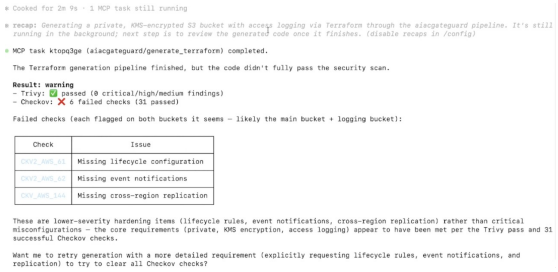
[2] `https://youtu.be/ea1dqkru_Tg`



(a) GitLab CI/CD pipeline #2704127824 triggered by the MCP server, completing in 14:34 with all Stages 1–5 passing.



(b) Natural-language instruction issued via Claude Code.



(c) Scanner results: Trivy ✓ (0 critical/high-/medium); Checkov ✗ 6 hardening checks flagged (CKV2_AWS_61, CKV2_AWS_62, CKV_AWS_144).

Figure 3. AIACGATEGUARD end-to-end MCP workflow.

Figure 3 illustrates a complete end-to-end interaction: the user issues a natural-language instruction via Claude Code (Figure 3b), the MCP server triggers the pipeline (Figure 3a), and the scanner results are returned directly in the chat (Figure 3c).

More specifically, the demonstration covers:

1. *Pipeline execution* (Figure 3a). Show the three declarative configuration files and trigger a reduced pipeline, monitoring the Stage 1–5 job matrix in real time.
2. *Result inspection* (Figure 3c). Walk through the per-model summary table, compliance heatmap, and a `summary.json` artifact, mapping a scanner finding to the offending Terraform resource.
3. *MCP conversational interface* (Figures 3b–3c). Using an MCP-compatible client, issue a natural-language instruction (*"create a private S3 bucket with KMS and logging"*), show `generate_terraform` returning the artifact with scanner results, then invoke `apply_terraform` to deploy to a sandbox account.

AIACGATEGUARD was previously validated through a case study evaluating seven models across 17 AWS Terraform scenarios, producing 3,570 automated evaluations. The results confirmed that syntactic validity and security compliance are largely independent properties of LLM-generated IaC. The complete experimental evaluation is presented in the companion paper [Vargas et al. 2026b].

5. Conclusion and Future Work

We presented AIACGATEGUARD, an open-source GitLab CI/CD pipeline that automates security compliance evaluation of LLM/SLM-generated Terraform by integrating dual-scanner analysis with Checkov and Trivy, optional AWS infrastructure stages, and an MCP server for conversational interaction. A case study involving seven models across

17 scenarios and 3,570 automated evaluations showed that syntactic validity and security compliance are largely independent properties, revealing a security dimension that existing IaC benchmarks focused primarily on deployability may overlook.

Beyond academic benchmarking, AIACGATEGUARD supports several practical applications. First, DevSecOps teams can use the pipeline to systematically identify and remediate security gaps in LLM-generated Terraform before production adoption. Second, security-tool and policy developers can use the benchmark to evaluate and improve scanner coverage against patterns commonly produced by AI coding tools. Third, the tool provides a practical environment for education, allowing students and practitioners to explore the intersection of AI-assisted coding, infrastructure-as-code security, and DevSecOps while observing security gaps that syntactic validation alone cannot detect.

The current implementation supports Terraform and AWS resources only. Extending coverage to additional IaC languages and cloud environments, such as CloudFormation, Pulumi, and Ansible, requires additional normalizer adapters. Stage 6 currently performs a single feedback round per invocation, meaning that multiple correction cycles must be manually triggered by the user. Wilson confidence intervals and Bonferroni-corrected chi-square significance tests are also provided as post-processing scripts rather than integrated into Stage 5. These statistical analyses require the complete cross-model result set, whereas Stage 5 executes independently for each model as soon as its corresponding generations are completed. Keeping the statistical analysis external also allows its methodology to evolve independently of the pinned pipeline code and Docker images. Automated integration of these analyses into Stage 5 is therefore left as future work.

AIACGATEGUARD is publicly available under an open-source license, including the complete pipeline implementation, prompt templates, scenario definitions, evaluation artifacts, and a video demonstration of installation and usage.[3]

Future work. Planned extensions include: (i) a web dashboard for interactive compliance analysis; (ii) plugin-based support for additional IaC languages, scanners, and policy-as-code frameworks such as OPA/Rego; (iii) optional multi-round automated revision with configurable stopping criteria while preserving the current human-in-the-loop default; (iv) a REST API with authentication and rate limiting for shared multi-user access; (v) integration of statistical post-processing into the evaluation pipeline; and (vi) a resource-efficient execution mode that runs SLMs sequentially for deployment on smaller hardware.

Acknowledgments

This work was partially supported by CAPES, CNPq (Grant No. 409743/2025-9), FAPERGS (Grant Nos. 24/2551-0001368-7 and 24/2551-0000726-1) and FAPESP (Grant Nos. 2020/05183-0 and 2023/00816-2). The authors also acknowledge the use of Generative AI as a supporting tool for reviewing and improving the code and manuscript.

[3] https://gitlab.com/security-iac/security-first-evaluation-of-text-to-terraform/-/tree/feat/sf-sbseg

References

Anthropic (2024). Model context protocol. `https://modelcontextprotocol.io`.

Aqua Security (2024). Trivy: Comprehensive security scanner. `https://trivy.dev/`.

Bridgecrew (2024). Checkov: Static code analysis for infrastructure-as-code. `https://www.checkov.io/`.

Brown, S. (2018). *Software Architecture for Developers*. Leanpub. C4 model: `https://c4model.com`.

Kon, P. T. J., Liu, J., Qiu, Y., Fan, W., He, T., Lin, L., Zhang, H., Park, O. M., Elengikal, G. S., Kang, Y., et al. (2024). IaC-Eval: A code generation benchmark for cloud infrastructure-as-code programs. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 37, pages 134488–134512.

Morris, K. (2020). *Infrastructure as Code: Dynamic Systems for the Cloud Age*. O'Reilly Media, 2nd edition.

Vargas, F. L. S., Mansilha, R. B., and Kreutz, D. (2026a). Can language models generate secure Terraform code? A security-focused benchmark using static analysis. In *Anais do I Simpósio de Infraestrutura Digital/Nuvem para Pesquisa (Pesquisa@Nuvem)*, pages 29–38. SBC.

Vargas, F. L. S., Mansilha, R. B., and Kreutz, D. (2026b). Security-first evaluation of text-to-Terraform: Benchmarking LLMs and SLMs for secure IaC generation. In *Anais do XXVI Simpósio Brasileiro de Segurança da Informação (SBSeg 2026)*. To be published.

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