From Intent to Deploy: Validator-Centric Multi-Agent Orchestration for Reliable Infrastructure-as-Code

Rana Nameer Hussain Khan, Dawood Wasif, Jin-Hee Cho, Ali R. Butt
Department of Computer Science
Virginia Tech
Blacksburg, VA, USA
e-mail: rnameerkhan@vt.edu, dawoodwasif@vt.edu, jicho@vt.edu, butta@vt.edu

***Abstract*—The increasing complexity of cloud-native infrastructure has made Infrastructure-as-Code (IaC) essential for reproducible and scalable deployments. While Large Language Models (LLMs) have shown promise in generating IaC snippets from natural language prompts, their monolithic, single-pass generation approach often results in syntactic errors, policy violations, and unscalable designs. In this paper, we propose MACOG (Multi-Agent Code-Orchestrated Generation), a novel multi-agent LLM-based architecture for IaC generation that decomposes the task into modular subtasks handled by specialized agents: Architect, Provider Harmonizer, Engineer, Reviewer, Security Prover, Cost and Capacity Planner, DevOps, and Memory Curator. The agents interact via a shared-blackboard, finite-state orchestrator layer, and collectively produce Terraform configurations that are not only syntactically valid but also policy-compliant, semantically coherent. To ensure infrastructure correctness and governance, we incorporate Terraform Plan for execution validation and Open Policy Agent (OPA) for customizable policy enforcement. We evaluate MACOG using the IaC-Eval benchmark, where MACOG is the top enhancement across models, e.g., GPT-5 improves from 54.90 (RAG) to 74.02 and Gemini-2.5 Pro from 43.56 to 60.13, with concurrent gains on BLEU, CodeBERTScore, and an LLM-judge metric. Ablations show constrained decoding and deploy feedback are critical. Removing them drops IaC-Eval to 64.89 and 56.93, respectively.**

Keywords-Infrastructure as Code; multi-agent systems; Large Language Models; program synthesis; policy as code.

I. INTRODUCTION

Cloud platforms expose a rapidly expanding surface of services, configuration parameters, and compliance regimes, making Infrastructure-as-Code (IaC) essential for reproducible deployment and operations [1]. Tools such as Terraform, Pulumi, and CloudFormation encode desired cloud state as declarative programs that must satisfy provider schemas, remain consistent across interdependent resources, and obey organizational constraints on security, cost, and data residency. Large language Models (LLMs) [2] promise to translate natural-language intent into IaC, yet practical correctness remains difficult. Deployments fail on version-sensitive schema details, fragile cross-resource references, and dependency structures that surface only during `plan/apply`. In production, IaC must also enforce least privilege, encryption, region pinning, redundancy targets, and budget limits. IaC synthesis is therefore constrained program construction validated by multiple independent authorities: schema checks, policy engines, cost estimators, and real toolchains.

These challenges are amplified by cloud evolution and modular reuse: provider APIs deprecate fields, migrate services across regions, and shift defaults, while modules are composed across teams and time with mismatched interfaces. IaC is inherently graph-structured, so near-miss generations often fail due to small but decisive mismatches (attributes, module wiring, region capabilities) that surface only under heterogeneous validators: security policies (OPA/Rego and scanners, such as Checkov/Regula), cost estimators, and real `plan/apply` toolchains. Prior work improves reliability through prompting, retrieval-augmented generation [0], and iterative refinement, but these approaches remain brittle under schema drift, multi-module dependencies, and unstable repair loops, and they rarely maintain a typed view of the infrastructure graph. Reliable assistants must therefore place validators at the center of the workflow rather than treat them as optional feedback.

We introduce *Multi-Agent Code-Orchestrated Generation* (MACOG), a validator-centric framework for generating deployable and auditable Terraform from natural-language intent without any model fine-tuning. MACOG (i) represents intent as a typed Infrastructure Intermediate Representation (I-IR) capturing resources, dependencies, regions, and required effects; (ii) compiles I-IR into Terraform using deterministic resource skeletons plus grammar- and schema-constrained decoding that stays within HCL [3] and provider field sets, enforced with a round-trip structural check; and (iii) closes the loop with tool-grounded validation and repair, combining `terraform validate`, policy enforcement (OPA/Rego with complementary scanners), deterministic cost estimation, and sandboxed `plan/apply` to produce structured counterexamples that drive minimal plan- or code-level edits. Agents coordinate over a versioned blackboard and reuse verified typed motifs rather than raw snippets to mitigate version drift. The result is a Terraform program paired with an evidence bundle that supports offline verification and audit; this validator-centric design also implies overhead and depends on the fidelity and availability of the underlying toolchains and sandboxes.

The remainder of this paper is organized as follows. Section II (§II) reviews related work on IaC generation, multi-agent code synthesis, and validator-guided repair. Section III (§III) presents MACOG, including the I-IR representation, blackboard orchestration, constrained compilation, and counterexample-driven repair. Section IV (§IV) describes the experimental setup, benchmarks, baselines, and metrics. Section V (§V) discusses

results and key trade-offs. Section VI (§VI) concludes and outlines future work.

II. Related Work

This section reviews prior work on IaC generation, multi-agent code synthesis, and validator-guided repair, which together motivate the design of MACOG.

A. LLMs for IaC and Configuration Synthesis

LLM-based IaC synthesis for Terraform [4], CloudFormation [5], and Ansible [6] maps natural-language intent to deployable artifacts under strict schemas and cross-resource references. Evidence from IaC-Eval shows one-shot prompting is brittle: even strong models omit required fields, misuse identifiers, and violate provider constraints without tool feedback [7]. Configuration-quality research further documents "smells" that degrade maintainability and security beyond syntax [9]. Security analyses catalog recurring IaC weaknesses (hard-coded secrets, overly permissive policies), motivating policy-aware validation [10], and large-scale measurements of Terraform practice reveal persistent gaps in adoption and enforcement [11]. Together, these results motivate systems that integrate schema awareness with policy compliance and runtime validation rather than relying only on prompting or retrieval.

B. Multi-Agent and Tool-Augmented Code Generation

A complementary line structures code generation as collaboration among specialized agents and external tools. *ChatDev* shows that role specialization, shared memory, and critique improve end-to-end correctness [0], while *self-collaboration* reports gains from plan–code–test role cycling within one model [12]. For software evolution, *MAGIS* coordinates planning and QA to resolve GitHub issues more reliably than single-agent prompting [13], and *RepairAgent* couples LLMs with a finite-state tool controller for diagnosis, patching, and validation [14]. These systems support orchestrations that route typed artifacts through schema, policy, and runtime validators while preserving already-correct structure.

C. Constrained Decoding and Validator-Guided Repair

Constrained decoding reduces invalid structured outputs by restricting generation to grammar- or schema-admissible tokens. *PICARD* enforces incremental parsing constraints for formal languages such as SQL [15]; *SynCode* applies DFA-style masking for CFGs to guarantee syntactic validity efficiently [16]; and *Grammar-Aligned Decoding* (ASAp) aligns sampling with the constrained conditional distribution to mitigate bias from hard constraints [17]. On the repair side, CEGIS uses counterexamples from verification to drive targeted refinements [18], and recent LLM repair work frames refinement as guided search over failures and partial successes [19]. For IaC, combining grammar/schema-constrained realization with validator-guided repair (static checks, policy engines, and sandboxed `plan/apply`) prevents invalid syntax upfront and turns machine-readable counterexamples into minimal, localized patches.

III. Methodology

This section describes a validator-centric pipeline that turns natural-language intent into deployable, secure, and cost-aware Terraform. The system is organized as a blackboard workflow with role-specialized agents operating over a typed Infrastructure Intermediate Representation (I-IR). It compiles I-IR into Terraform using grammar- and schema-constrained decoding, then iteratively repairs failures using structured counterexamples produced by authoritative external validators. The workflow runs entirely in instruction-following mode with no model fine-tuning, and it enforces correctness through deterministic tooling, explicit typing, and evidence-driven edits.

A. Problem Setup

Input consists of a natural-language intent x and a constraint set C that specifies operational requirements, such as budget, residency, encryption, and availability. Output is a Terraform program T together with an evidence bundle Π that supports independent verification.

a) Typed plan representation (I-IR): We represent the intended infrastructure as a typed resource graph $P = \langle V, E, S \rangle$:

- V are resource nodes with provider-specific types and typed fields.
- E are dependency and connectivity edges that make ordering and wiring explicit.
- S are specifications and effects, including security and compliance obligations.

I-IR makes the infrastructure graph first-class: resources, edges, regions, and obligations are explicit and mechanically checkable.

b) Validation contracts: A candidate Terraform program must satisfy four contracts:

- **Schema validity:** provider schemas and references are correct (`terraform validate` and schema matchers).
- **Policy compliance:** organization rules hold (OPA/Rego plus complementary scanners).
- **Cost feasibility:** cost is within the stated budget using pinned catalogs and line-item summaries.
- **Deployability:** `init/plan/apply` succeeds in controlled sandboxes.

The system treats validator outputs as ground truth and uses them to drive precise repairs.

B. System Overview

MACOG uses a blackboard architecture: each agent reads and writes structured artifacts (typed plans, diffs, diagnostics, logs) and the orchestrator advances a fixed state machine. This design prevents uncontrolled rewrites, preserves correct structure across iterations, and makes every change traceable. As shown in Figure 1, MACOG coordinates agents over a shared blackboard and uses validator feedback.

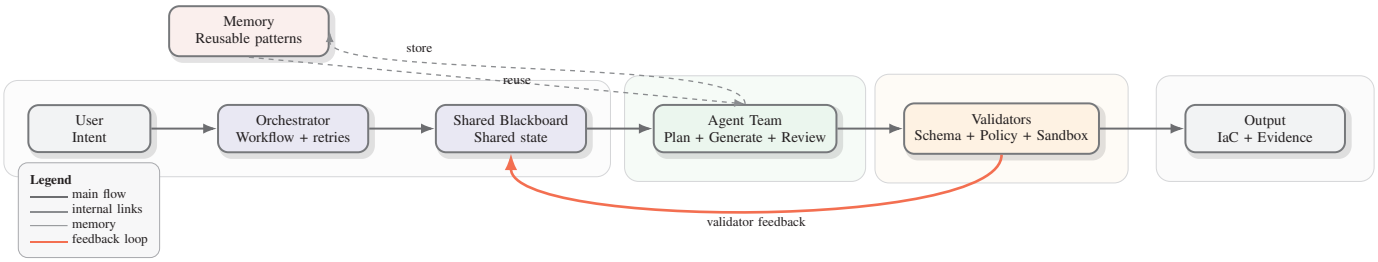

Figure 1. High-level MACOG architecture.

a) Roles and responsibilities:

- **Architect:** converts (x, C) into an initial I-IR plan P_0 and a set of invariants $\mathcal{I}$ that explicitly capture obligations such as encryption, residency, and exposure bounds.
- **Provider Harmonizer:** resolves provider versions, regions, and required fields, producing a fully concrete plan P_1 consistent with pinned schemas.
- **Engineer:** compiles P_1 into Terraform using constrained decoding that enforces HCL grammar and provider field admissibility.
- **Reviewer:** runs static checks (`terraform validate`, linters, module interface checks) and produces structured, localized diagnostics.
- **Security Prover:** executes OPA/Rego and complementary scanners and returns policy traces and counterexample witnesses.
- **Cost & Capacity Planner:** produces deterministic cost sheets from pinned catalogs and flags SKU, quota, and region feasibility issues.
- **DevOps:** runs `init/plan/apply` in sandboxes (LocalStack for fast feedback, ephemeral accounts for realism) and returns runtime error objects and logs.
- **Memory Curator:** stores verified motifs as typed I-IR fragments with schema/version metadata and serves them during planning and compilation.

C. I-IR: Typed Graph with Explicit Obligations

Each resource node carries typed fields and explicit effects:

- **Fields** are typed by pinned provider schemas, so required attributes and admissible values are enforced at the plan level.
- **Effects** encode obligations such as encryption-at-rest and least-privilege, which are later discharged by policy validators.
- **Edges** encode ordering and connectivity, which eliminates hidden dependencies and stabilizes multi-module generation.

This representation eliminates ambiguity: the system always operates on a concrete, checkable infrastructure graph.

D. Constrained Compilation with Round-Trip Checks

Compilation is intentionally strict and proceeds in two stages:

a) Stage 1: Structural skeleton: A structural compiler maps P_1 to a Terraform skeleton $\widetilde{T}$:

- required blocks are emitted first,
- module boundaries, variables, and outputs are constructed deterministically,
- references are filled from a symbol table derived from node identifiers.

b) Stage 2: Constrained decoding: The Engineer completes remaining fields using constrained decoding:

- HCL grammar constraints prevent syntactically invalid output,
- provider-schema constraints prevent illegal fields and incompatible values,
- reference constraints enforce correct attribute wiring and scope.

c) Round-trip equivalence: Before any expensive checks, the system parses the generated Terraform back into a plan and enforces structural equivalence with P_1 modulo harmless normalization (field ordering and renaming). This guarantees that compilation preserves intent and prevents drift across iterations.

E. Validator-Grounded Repair

After compilation, MACOG runs the full validator suite and collects structured counterexamples:

- **Schema counterexamples:** missing required attributes, type mismatches, broken references.
- **Policy counterexamples:** rule identifiers, failed predicates, and witnesses from OPA/Rego and cross-check scanners.
- **Cost counterexamples:** line items exceeding budget, region pricing mismatches, capacity flags.
- **Runtime counterexamples:** `plan/apply` failures including dependency cycles, unsupported SKUs, quota errors, and missing identifiers.

a) Deterministic error-to-edit mapping: Counterexamples are mapped to minimal edits using a fixed set of operators:

- **Plan-level edits** modify I-IR when failures are structural (region, topology, connectivity, required effects).
- **Code-level edits** modify Terraform when failures are local (single field, reference, block attribute).

Edits are applied with a strict priority order: structural fixes first, then localized patching. This produces fast convergence and prevents late-stage thrashing.

TABLE I. AVERAGE IAC-EVAL TASK SUCCESS (%, HIGHER IS BETTER) ACROSS MODELS UNDER FIVE ENHANCEMENT STRATEGIES.

Rank	Name	Few-shot	CoT	Multi-turn	RAG	MACOG
1	GPT-5	12.53	10.19	35.83	54.90	74.02
2	Gemini-2.5 Pro	12.18	10.49	36.81	43.56	60.13
3	GPT-4	10.64	9.31	31.12	36.70	43.20
4	GPT-3.5-turbo	0.80	1.60	11.44	21.81	25.40
5	Gemini 2.0 Flash	3.33	1.80	4.93	10.32	17.85
6	Magicoder-S-CL-7B	2.93	0.53	12.50	12.77	16.95
7	WizardCoder-33B-V1.1	1.60	1.06	9.04	11.70	15.80
8	CodeLlama Instruct (34B)	3.19	3.19	2.13	6.12	10.45
9	CodeLlama Instruct (7B)	2.39	3.72	0.53	5.59	9.70
10	CodeLlama Instruct (13B)	1.06	1.86	1.06	3.46	6.40

b) Routing objective: The orchestrator uses a single compact score to prioritize which failures to address first:

$$J(T,C) = \lambda_s\,(1 - v_{\text{schema}}) + \lambda_p\,(1 - v_{\text{policy}}) + \lambda_d\,(1 - v_{\text{deploy}}) + \lambda_c\,\max(0, \widehat{\text{cost}}(T) - B), \quad (1)$$

where $v_{\text{schema}}, v_{\text{policy}}, v_{\text{deploy}} \in \{0,1\}$ are validator pass indicators, $\widehat{\text{cost}}(T)$ is the deterministic cost estimate, and B is the budget. This score is used only for deterministic control flow and never as a learned loss.

F. Blackboard Orchestration and Reproducibility

All artifacts are written to a typed blackboard: I-IR versions, Terraform candidates, diffs, validator traces, cost sheets, and deploy logs. Every entry is stamped with provider schema versions, toolchain digests, and content hashes.

a) Finite-state orchestration: The orchestrator advances a fixed state machine with the following states:

$$\mathsf{S} \in \left\{ \begin{array}{l} \texttt{plan, harmonize, compile, review,} \\ \texttt{prove, price, deploy, repair, done} \end{array} \right\}.$$

Transitions are guarded by explicit contracts, expressed as preconditions over the blackboard artifacts (e.g., `schema-valid`, `policy-valid`, `budget-valid`, `deploy-valid`) in the sense of Design by Contract [0]. This makes behavior predictable, prevents uncontrolled rewrites, and keeps failures fully diagnosable from the recorded traces.

G. Tooling and Proof-Carrying Output

MACOG returns (T, Π):

- T is the final Terraform program.
- Π is a self-contained evidence bundle with policy traces and witnesses, cost sheets with catalog versions, schema snapshots, static validation logs, deploy logs, round-trip equivalence records, and the complete repair trajectory.

This output is audit-ready: the same validators can replay Π offline and confirm compliance before production execution.

IV. EXPERIMENTS

We evaluate MACOG on IaC-Eval [7], comparing it against four commonly used enhancement strategies (Few-shot, CoT, Multi-turn repair, and RAG) across a broad set of code-capable LLMs. We report a cross-model summary on harness-verified task success (IaC-Eval), then provide multi-metric profiles for two frontier models (GPT-5 and Gemini-2.5 Pro), and finally isolate MACOG's key components via ablations. All runs use the same task prompts, the same retry budget, and no fine-tuning.

TABLE II. GPT-5 UNDER FIVE ENHANCEMENT STRATEGIES ACROSS FOUR METRICS (0–100, HIGHER IS BETTER).

Strategy	BLEU	CodeBERTScore	LLM-judge	IaC-Eval
Few-shot	5.68	72.41	68.22	12.53
CoT	3.37	70.85	60.31	10.19
Multi-turn	5.54	71.08	62.17	35.83
RAG	10.71	76.43	69.72	54.90
MACOG	**11.86**	**80.54**	**94.10**	**74.02**

TABLE III. GEMINI-2.5 PRO UNDER FIVE ENHANCEMENT STRATEGIES ACROSS FOUR METRICS (0–100, HIGHER IS BETTER).

Strategy	BLEU	CodeBERTScore	LLM-judge	IaC-Eval
Few-shot	5.12	65.08	57.41	12.18
CoT	4.94	61.77	56.20	10.49
Multi-turn	8.87	66.95	58.03	36.81
RAG	9.73	69.92	64.15	43.56
MACOG	**10.09**	**71.84**	**87.52**	**60.13**

A. Setup and Protocol

a) Benchmark: IaC-Eval [7] provides natural-language infrastructure intents with verification procedures designed for Terraform-based provisioning. Tasks span networking, compute, storage, IAM, and managed services, and require globally consistent cross-resource references and provider-valid schemas. A task is counted as solved only if it passes the benchmark harness checks.

b) Models: We evaluate closed and open models representative of modern IaC generation: GPT-5, GPT-4, GPT-3.5-turbo [20]; Gemini-2.5 Pro and Gemini 2.0 Flash [21]; and open-weight code models (Magicoder-S-CL-7B, WizardCoder-33B, CodeLlama Instruct 7B/13B/34B). The same strategy logic is applied across models to attribute gains to the strategy rather than model-specific prompt tuning.

c) Strategies: We compare five increasingly structured approaches: **(i) Few-shot** single-turn exemplars, no tools; **(ii) CoT** single-turn with explicit reasoning before code, no tools; **(iii) Multi-turn** bounded retries with validator feedback paraphrased in natural language; **(iv) RAG** retrieval

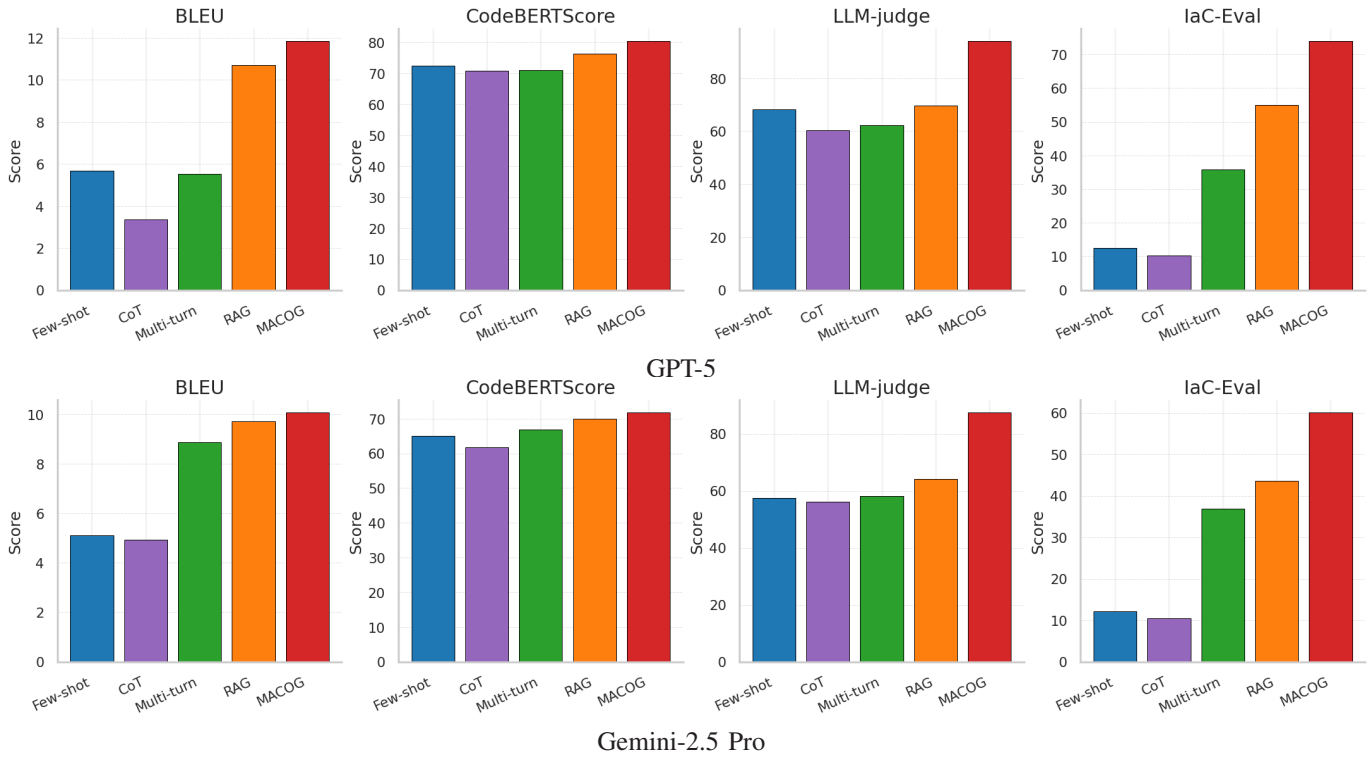

Figure 2. Metric profiles across enhancement strategies (BLEU, CodeBERTScore, LLM-judge, IaC-Eval).

TABLE IV. ABLATION STUDY OF MACOG COMPONENTS ON GPT-5 (0–100, HIGHER IS BETTER).

Variant	BLEU	CodeBERT	Judge	IaC-Eval
Full MACOG	**11.86**	**80.54**	**94.10**	**74.02**
– Harmonizer	10.98	78.92	92.48	70.37
– Engineer (no constr. dec.)	8.61	73.15	89.74	64.89
– Reviewer	10.27	76.04	86.11	66.72
– Security Prover	10.81	77.53	90.03	61.45
– Cost & Capacity	11.22	79.38	92.01	71.08
– DevOps (no sandbox)	9.47	74.82	88.57	56.93
– Memory Curator	10.95	79.06	91.34	72.17

of similar solved tasks as short sanitized hints, no hard constraints; **(v) MACOG** multi-agent orchestration over typed I-IR, grammar/schema-constrained realization, round-trip structural checks, and validator-driven counterexample repairs.

d) Metrics: We report BLEU, CodeBERTScore–F1, LLM-judge, and IaC-Eval using the official IaC-Eval implementations in the code repository [7].

e) Toolchain and validators: All runs use a pinned Terraform toolchain for static validation, provider schema snapshots for required-field/type checks, OPA/Rego policies with a curated ruleset plus a second scanner as a cross-check, and a deployment sandbox (LocalStack and ephemeral accounts) where permitted. Validator outputs are stored as structured JSON traces. Only MACOG consumes structured traces programmatically; Multi-turn receives a natural-language summary; Few-shot/CoT/RAG receive no validator traces.

f) Artifacts and reproducibility: We use the public IaC-Eval benchmark [7], available at https://huggingface.co/datasets/autoiac-project/iac-eval. All code and evaluation scripts are archived on Zenodo at https://zenodo.org/records/17117489.

B. Results

a) Frontier model profiles: Figure 2 and Table II–Table III show that MACOG improves not only IaC-Eval success but also judged adequacy and semantic similarity. For GPT-5, MACOG yields simultaneous gains in BLEU, CodeBERTScore, LLM-judge, and IaC-Eval, with the largest jump in LLM-judge and IaC-Eval, indicating that validator-grounded repairs systematically convert near-miss configurations into deployable and policy-consistent programs. Gemini-2.5 Pro shows the same pattern with a strong uplift in LLM-judge and IaC-Eval, reflecting improved adherence to security and correctness constraints under the same prompt budget.

b) Ablation study: Table IV shows that the main gains come from runtime grounding, policy grounding, and constrained decoding: removing the DevOps sandbox yields the largest IaC-Eval drop ($74.02 \rightarrow 56.93$), followed by removing the Security Prover ($74.02 \rightarrow 61.45$) and disabling constrained decoding ($74.02 \rightarrow 64.89$). The remaining modules provide smaller but consistent gains by aligning plans to schemas early and catching interface errors cheaply.

V. DISCUSSION

The experiments show that IaC generation improves most when correctness is defined and enforced by validators rather

than prompt-only fluency: MACOG achieves the highest IaC-Eval success across models (Table I) while also improving adequacy (LLM-judge) and semantic similarity (CodeBERTScore) for GPT-5 and Gemini-2.5 Pro (Table II, Table III). These gains follow from the design: typed planning and grammar/schema-constrained realization prevent many structural and field-level errors by construction, and structured counterexamples from schema, policy, cost, and deployment validators drive localized repairs instead of broad rewrites, which stabilizes convergence under real toolchains. Ablations support this mechanism: removing the DevOps sandbox yields the largest drop in success (Table IV), indicating runtime-grounded feedback is decisive for last-mile failures that static checks miss, while removing the Security Prover and constrained decoding produces the next largest degradations, highlighting the value of policy witnesses and admissible decoding. The remaining modules contribute steady improvements by aligning plans with provider realities and catching cheap interface defects early, and the same principles extend beyond Terraform by swapping the compiler backend and validator suite. In terms of overhead, MACOG trades single-shot latency for higher acceptance by paying additional runtime for multi-agent coordination and repeated validator execution, with `plan/apply` typically dominating end-to-end time.

VI. Conclusion and Future Work

This paper presented MACOG, a multi-agent, validator-centric pipeline that synthesizes deployable and compliant Infrastructure-as-Code from natural-language intent without fine-tuning. MACOG uses a typed infrastructure intermediate representation to preserve global structure, grammar- and schema-constrained compilation with round-trip checks to prevent drift, and counterexample-guided repair grounded in static validation, policy enforcement, cost estimation, and sandboxed `plan/apply`. Across ten models and five enhancement strategies, MACOG achieves the highest harness-verified task success on IaC-Eval and improves adequacy and semantic similarity for frontier models (Table I, Table II, Table III); ablations confirm that constrained realization, policy grounding, and runtime grounding account for most gains (Table IV).

Future work will reduce latency via parallel validation, caching, and incremental re-checks, and improve robustness to provider evolution through automated schema/catalog refresh and drift-triggered re-harmonization. Additional directions include richer policies (e.g., residency and supply-chain) with stronger witnesses, multi-cloud backends via swappable compiler/validator suites, and replayable evidence bundles for offline audit and CI/CD gating.

References

[1] A. Rahman, R. Mahdavi-Hezaveh, and L. Williams, "A systematic mapping study of infrastructure as code research", *Information and Software Technology*, vol. 108, pp. 65–77, 2019.

[2] Y. Chang et al., "A survey on evaluation of large language models", *ACM transactions on intelligent systems and technology*, vol. 15, no. 3, pp. 1–45, 2024.

[0] P. Lewis et al., "Retrieval-augmented generation for knowledge-intensive nlp tasks", *Advances in neural information processing systems*, vol. 33, pp. 9459–9474, 2020.

[3] P. Riti and D. Flynn, *Beginning HCL Programming*. Springer, 2021.

[4] K. Shirinkin, *Getting Started with Terraform*. Packt Publishing Ltd, 2017.

[5] K. Tovmasyan, *Mastering AWS CloudFormation: Plan, develop, and deploy your cloud infrastructure effectively using AWS CloudFormation*. Packt Publishing Ltd, 2020.

[6] J. McAllister, *Implementing DevOps with Ansible 2*. Packt Publishing Ltd, 2017.

[7] 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.

[9] J. Schwarz, A. Steffens, and H. Lichter, "Code smells in infrastructure as code", in *2018 11Th international conference on the quality of information and communications technology (QUATIC)*, IEEE, 2018, pp. 220–228.

[10] L. C. Opara, O. N. Akatakpo, I. C. Ironuru, K. Anyaene, and B. O. Enobakhare, "Chaos engineering 2.0: A review of ai-driven, policy-guided resilience for multi-cloud systems", *Journal of Computer, Software, and Program*, vol. 2, no. 2, pp. 10–24, 2025.

[11] A. Verdet, M. Hamdaqa, L. D. Silva, and F. Khomh, "Exploring security practices in infrastructure as code: An empirical study", *arXiv preprint arXiv:2308.03952*, 2023. arXiv: 2308.03952 [cs.CR].

[0] C. Qian et al., "Chatdev: Communicative agents for software development", in *Proceedings of the 62nd annual meeting of the association for computational linguistics (volume 1: Long papers)*, 2024, pp. 15 174–15 186.

[12] Y. Dong, X. Jiang, Z. Jin, and G. Li, "Self-collaboration code generation via chatgpt", *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 7, pp. 1–38, 2024.

[13] W. Tao et al., "Magis: Llm-based multi-agent framework for github issue resolution", *Advances in Neural Information Processing Systems*, vol. 37, pp. 51 963–51 993, 2024.

[14] I. Bouzenia, P. Devanbu, and M. Pradel, "Repairagent: An autonomous, llm-based agent for program repair", *arXiv preprint arXiv:2403.17134*, 2024.

[15] T. Scholak, N. Schucher, and D. Bahdanau, "Picard: Parsing incrementally for constrained auto-regressive decoding from language models", *arXiv preprint arXiv:2109.05093*, 2021.

[16] S. Ugare, T. Suresh, H. Kang, S. Misailovic, and G. Singh, "Syncode: Llm generation with grammar augmentation", *Transactions on Machine Learning Research*, 2024.

[17] K. Park, J. Wang, T. Berg-Kirkpatrick, N. Polikarpova, and L. D'Antoni, "Grammar-aligned decoding", *Advances in Neural Information Processing Systems*, vol. 37, pp. 24 547–24 568, 2024.

[18] A. Solar-Lezama, "The sketching approach to program synthesis", in *Asian symposium on programming languages and systems*, Springer, 2009, pp. 4–13.

[19] H. Tang et al., "Code repair with llms gives an exploration-exploitation tradeoff", *Advances in Neural Information Processing Systems*, vol. 37, pp. 117 954–117 996, 2024.

[0] B. Meyer, "Applying "design by contract"", *Computer*, vol. 25, no. 10, pp. 40–51, 1992. DOI: 10.1109/2.161279.

[20] OpenAI, "Introducing `chatgpt`", Nov. 2022, Accessed: Nov. 15, 2025. [Online]. Available: https://openai.com/index/chatgpt/.

[21] G. Team et al., "Gemini: A family of highly capable multimodal models", *arXiv preprint arXiv:2312.11805*, 2023.