FROM RULE DIAGNOSTICS TO GUARDED COMPILATION FOR INFRASTRUCTURE-AS-CODE REPAIR

Anonymous authors
Paper under double-blind review

ABSTRACT

Repairing an infrastructure rule violation can require constructing a resource that is itself subject to further rules. We use this rule structure to compile reusable resource obligations and execute them under a structural guard before model revision. Our diagnostic study shows why both steps matter: adding resource locations to feedback raises scanner acceptance by 8.8 percentage points, concentrated on bucket-bearing templates, while joint scanner-and-lint acceptance stays flat. The intervention study evaluates a frozen compiler on 153 matched CloudFormation tasks. With Haiku 4.5 and a six-call refinement cap, guarded compilation reaches 88.2% joint acceptance versus Reflexion's 80.4%, using 1.78 versus 3.42 calls and 120.4k versus 230.1k input tokens per task. Executing the library adds 4.6 points over supplying it as context. Rule diagnostics thus motivate a concrete intervention: resolve supported obligations with checked code and reserve model revision for the residual failures. The benchmark was used during development, and the acceptance advantage does not persist with the tested stronger runtimes.

1 INTRODUCTION

Infrastructure-as-code (IaC) turns application requirements into resources, dependencies, and permissions. Generated code can violate security requirements (Pearce et al., 2022; Perry et al., 2023), and self-repair without reliable external feedback remains limited (Olausson et al., 2023). Even a local IaC finding can require a connected resource: enabling bucket logging needs a valid destination and permissions, not merely a nonempty property.

Checker-guided iteration is an established response (Madaan et al., 2023; Shinn et al., 2023; Palavalli & Santolucito, 2024). Yet richer feedback can improve the scanner score while leaving structural errors intact. Our location ablation makes this concrete: the gain concentrates on templates carrying S3 buckets, whose logging rule can trigger another obligation on a newly created logging bucket. A self-reference closes the scanner's presence check while creating a CloudFormation dependency cycle.

This observation suggests the organizing principle of the paper: *use rule structure to identify supported resource obligations, compile their construction, and check the edit before asking the model for another revision*. Our compiler prepares a frozen metadata-driven library with authored mappings. The runtime executes its supported constructions under an edit guard, then sends residual findings to the model (Figure 1). The paper connects diagnosis, rule structure, and intervention through three contributions:

1. **Diagnosis: localizing a finding and repairing it are different tasks** (§5). Locations raise scanner acceptance by 8.8 points but leave joint acceptance unchanged. Rule and resource strata identify where this difference arises.
2. **Mechanism and intervention: compile supported obligations** (§3). Rule metadata and schemas define reusable constructors; a guard preserves protected values and rejects structural regressions.
3. **Higher joint acceptance with fewer runtime tokens** (§6). The matched Haiku comparison gains 7.8 points over Reflexion while using 120.4k versus 230.1k input tokens per task. A fixed-prompt ablation measures a 4.6-point increment from executing the library.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

The resulting workflow reuses supported constructions across tasks and reserves model revision for the remaining failures.

2 Related Work

Feedback-driven generation and repair. Self-Refine uses self-feedback, while Reflexion turns task feedback into verbal reflections (Madaan et al., 2023; Shinn et al., 2023). Self-Debugging studies code explanation and execution feedback (Chen et al., 2024), and monitors constrain generation using program context (Agrawal et al., 2023). Olausson et al. (2023) examine the limits of self-repair. FeedbackEval varies the information supplied to repair models (Dai et al., 2025). These studies motivate controlling both feedback and the loop: our component comparison fixes the revision procedure while changing library availability and execution. Learning-based approaches instead adapt the generator: InferFix combines fine-tuning with retrieved fixes (Jin et al., 2023), and RLEF trains models to use execution feedback (Gehring et al., 2024). Our generator weights remain fixed.

Localization within an evolving artifact. Fault localization is a longstanding repair stage (Wong et al., 2016). ITER re-runs localization after partial patches to support multi-location repair (Ye & Monperrus, 2023). Recent work measures the effects of file-, element-, and line-level context (Sepidband et al., 2026), predicts suspicious lines for refinement (Yao et al., 2026), and compares line-anchored with holistic editing feedback (Lamberti, 2026). Perturbed-location experiments also probe vulnerability repair (Camporese & Massacci, 2025). We isolate resource identity in a fixed diagnostic renderer, then stratify its effect by the resource obligations induced by the checker. This connects feedback design to a specific source of ambiguity: creating a resource that is itself subject to the rule being repaired.

Reusable repair and constrained generation. Template-based repair already reuses fix patterns; TBar studies their interaction with localization and donor-code retrieval (Liu et al., 2019). InfraFix provides a technology-agnostic IaC repair framework, including an SMT-based repair implementation guided by inferred system state (Saavedra et al., 2025). PICARD and guided generation constrain token choices during decoding (Scholak et al., 2021; Willard & Louf, 2023). Our design specializes scanner metadata and resource schemas into constructors that insert and bind supported resources between model calls. The distinctive evaluation separates supplying those constructors as context from executing them under a structural guard. The contribution is this rule-to-construction interface and its measured execution increment, rather than reuse of repair templates alone.

IaC generation, policy compliance, and validity. Prior systems feed CloudFormation lint diagnostics back to the model (Palavalli & Santolucito, 2024), retrieve configuration knowledge (Nekrasov et al., 2025), or combine LLMs with policy engines (Romeo et al., 2025; Shin et al., 2026). TerraRepair retrieves dependency and provider-schema information within a bounded, scanner-verified repair agent (Mengistu et al., 2026); this is especially close to our runtime setting. TerraFormer fine-tunes models with policy-guided verifier feedback (Jana et al., 2026), while IaCGen uses DevOps simulation for deployability-oriented generation (Zhang et al., 2026). Security degradation during repair and scanner-passing deceptive fixes are already documented (Agyekum & Santos, 2026; Alsaid et al., 2026). IaC-Guard-V evaluates validity, issue resolution, regression safety, and patch minimality (Chauhan, 2026); Jouini (2026) separates Terraform validation, planning, and policy checks in agent evaluation. Our diagnostic contribution identifies a same-type resource obligation associated with the scanner–validity gap; the intervention tests whether executing compiled constructors adds value beyond exposing the same library to the model.

3 From Rule Structure to Guarded Execution

The method moves repeated resource-construction work into a reusable library. *Compilation* produces the constructors; *execution* instantiates and binds them in a candidate under a structural guard. The runtime rechecks the edited candidate and sends remaining failures to the model, repeating until joint acceptance or the call cap.

Figure 1 separates preparation from execution. Only generation and revision require runtime model calls. Compiled edits, structural guards, and feedback extraction run as code; verifiers inspect the candidate. Generation stops when scanner and structural checks pass.

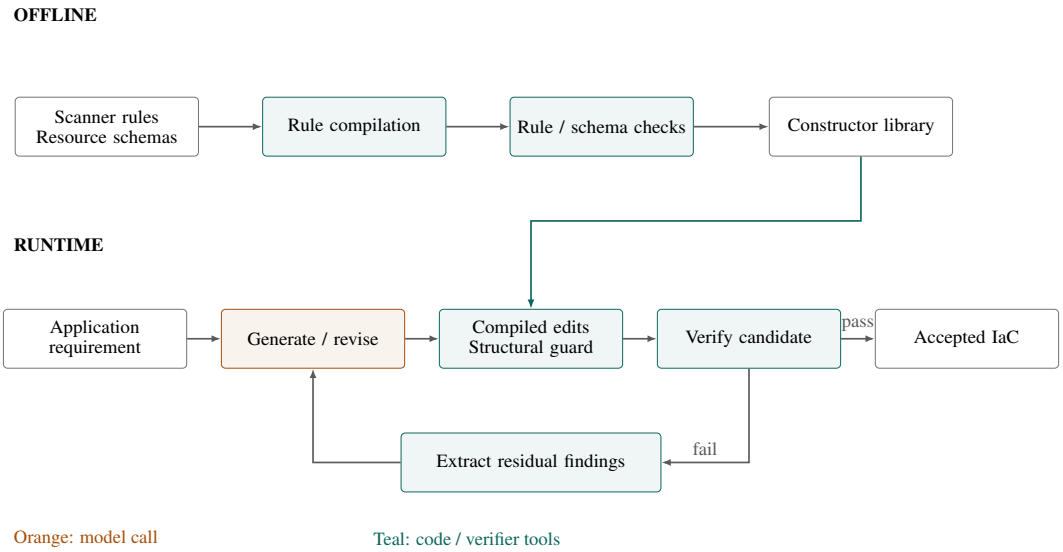


Figure 1: **The static compilation and execution procedure.** The metadata-driven, hand-assisted library is frozen and reused across requirements. At runtime, the guard checks compiled edits and failed joint verification returns residual findings to the model. The call budget covers the orange generation/revision node. Failed verification triggers another revision while budget remains.

3.1 Rule compilation

Raw rules describe acceptance conditions, leaving edit selection, schema completion, and resource binding to the runtime. A `KmsKeyId` presence check, for example, supplies neither a key resource nor a valid binding. An LLM can interpret such rules; compilation aims to avoid repeating this construction reasoning for each candidate. The compiler extracts inspected paths and expected values, maps reference-shaped paths to resource types through an authored lexicon, and builds schema-based constructor skeletons. Bounded rechecking completes supported values and dependencies; authored contracts delimit edits and exclude permission and network intent. The primary experiment uses this frozen metadata-driven, hand-assisted library, with six constructor families (App. B, C). Compilation itself uses no model proposals.

At runtime, edits must preserve property types, intrinsic expressions, and existing policy containers. A structural check rejects constructions that introduce new lint diagnostics; unsupported edits remain for model refinement. The guard compares diagnostics before and after the edit and returns the original candidate when a protected path or structural check fails. App. B gives the contracts and B.4 the runtime-call accounting.

3.2 Runtime generation

Supported edits can resolve findings or introduce new ones, so the runtime rechecks before constructing the next model input. It reconciles scanner and permission findings, selects blocking residuals, and retains the rule, severity, message, and resource/property location. The runtime also supplies parser and lint diagnostics. The model receives these diagnostics, the revised candidate, and the original requirement. The feedback representation keeps each blocking finding with its target resource or property. Section 6.3 compares it with native reports, and App. C illustrates the transformation.

Constructing dependent resources. Suppose `AppBucket` needs access logging. Setting its destination to `!Ref AppBucket` can pass the presence check yet create a circular CloudFormation dependency. Adding `LogBucket` avoids self-reference, but that bucket also needs logging. This is a *resource obligation*: satisfying a property requires another resource. The runtime constructs

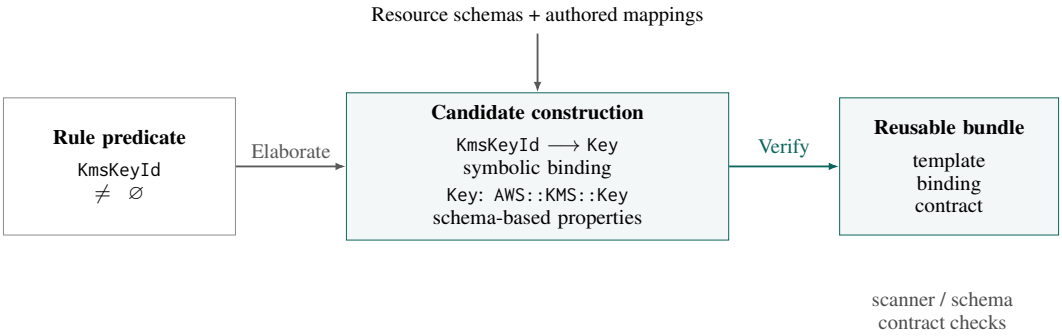


Figure 2: **Compilation turns a predicate into a reusable construction.** In this schematic KMS example, the predicate supplies neither a key resource nor its binding. Schemas and authored mappings support their construction; verification records remaining findings. The runtime binds the frozen bundle to a candidate and checks the edit.

`LogBucket`, binds its logging destination to a parameter naming a distinct existing bucket, then points `AppBucket` to `LogBucket`. The caller supplies that external destination and its permissions. The dependency graph orders supported constructions; finite bundles or external inputs end chains, with unsupported obligations left for refinement. New resources must pass supported checks, and internal references must resolve. Termination and preservation are conditional on the constructor contracts. App. B.2 formalizes the dependency semantics and construction contracts; the runtime guard checks the resulting artifact for new structural errors.

What the obligation graph captures. Write $t \xrightarrow{p} u$ when a supported property p of resource type t calls for a referent of type u. This typed graph combines inspected rule paths with schemas and authored reference mappings. The checker may require only a nonempty property (*literal acceptance*); the construction additionally supplies a typed referent. For logging, `S3::Bucket` $\to$ `S3::Bucket` exposes a possible cascade: a fresh witness is another bearer of the same rule. For log-group encryption, `Logs::LogGroup` $\to$ `KMS::Key` does not repeat that rule. The same-type edge makes self-reference available, not necessary. A finite logging bundle ends at an explicit external parameter; a cyclic obligation without a supported boundary is refused. These graph edges describe supported constructor obligations, not a decision procedure for arbitrary custom checker code.

A checkable commit rule. Let x' be a proposed construction on x, $P(x, x')$ denote the protected-path checks, and $L(x)$ be the multiset of lint diagnostics indexed by rule and property path. The guard requires $P(x, x') \wedge L(x') \setminus L(x) = \varnothing$ before committing the edit; a failed verifier run also rejects it. Multiset comparison retains duplicate counts, so moving a finding to a new path or adding another instance cannot masquerade as preservation. The protected checks reject replacement of existing containers or intrinsic expressions and changes to existing policy blocks. Thus a scanner-passing intrinsic self-reference is rejected when it adds an E3004 cycle. This is a local regression guard: existing errors remain residuals for refinement, and external resource existence and permissions remain caller obligations.

4 Experiments

4.1 Tasks and study roles

Both studies use the 153 CloudFormation requirements from DPIaC-Eval (Zhang et al., 2026). The *diagnostic study* compares feedback with and without resource locations, with compilation disabled, five shared draws per requirement (765 cells), and at most $B = 3$ additional revision calls. It stops at scanner acceptance and is rescored for lint and AWS validation. The *intervention study* uses one prospectively selected shared Haiku draft per requirement (153 cells), $B = 6$, and joint stopping. Its byte-identical initial drafts are reused across methods. The tasks were inspected during development.

4.2 Common experimental protocol

In the intervention study, all methods receive the requirements, scanner rules, schemas, and the same initial drafts. The frozen metadata-driven library and guarded compiler are identical in the main comparison, component study, and runtime-model transfer. The main comparison and component study use Haiku 4.5 for refinement; transfer changes only the runtime model to Opus 5.5 or Fable 5.1. Provider identifiers and prompt builders are specified in App. A.

The cap is $B = 6$ additional model calls, excluding shared initial generation. Critique, reflection, explanation, and agent tool-selection calls all consume the cap; each response permits at most $16{,}000$ output tokens. All methods stop on joint scanner-and-lint acceptance. Checkov 3.2.403 and the fixed IAM rule provide scanner feedback; cfn-lint 1.55.1 provides structural diagnostics. The budget study varies $B = 0, \ldots, 6$ under this same protocol.

4.3 Baselines

We adapt Self-Refine (Madaan et al., 2023), Reflexion (Shinn et al., 2023), and Self-Debugging (Chen et al., 2024) to IaC, preserving their critique–revision, reflection–revision, and explanation–revision stages, respectively. Reflexion uses the latest reflection; Self-Debugging retains prior turns. Claude Code (Anthropic, 2026) receives file/shell tools and the scanner; a proxy enforces its call cap. Baselines use raw rule/schema context; our method uses the compiled library. Tool-guided methods share extracted findings; Self-Refine uses the verifier only for stopping. The direct feedback control shares our revision prompt; native Checkov JSON plus IAM findings controls report representation. The component study fixes the revision prompt and changes compiled context, execution, or feedback representation one at a time.

4.4 Outcomes, cost, and inference

Table 1 defines the outcomes used throughout the paper. Intervention rates retain all 153 tasks; diagnostic rates retain all 765 cells. Unknown quality outcomes remain unconfirmed, and missing token usage is marked explicitly; App. A.1 gives the outcome bounds.

Measure	Definition
Scanner pass	Parseable template; no blocking findings under the fixed Checkov/IAM gate; minimum resource-count guard
Lint pass	Parseable template; no cfn-lint E findings on canonical JSON in `us-east-1`
Joint pass	Scanner and lint both pass on the same final artifact
New calls; input/output tokens	Successful additional responses and recorded token usage per scheduled task; shared initial generation and offline compilation excluded

Table 1: **Intervention outcomes and cost definitions.** Input tokens include uncached input, cache creation and cache reads. Missing usage produces a marked lower bound rather than an imputed zero.

We report 95% Wilson intervals for marginal acceptance rates and paired requirement-bootstrap intervals for differences. App. A.3 specifies the resampling procedures. The budget figure reports both the controlled call cap and observed token expenditure.

5 Diagnosis: Locations, Rule Obligations, and Validity

Locations help the scanner criterion. Adding resource/property locations to otherwise identical feedback raises scanner acceptance from 61.6% to 70.3% ($+8.8$pt). Joint scanner-and-lint acceptance stays at 20.9% versus 20.8%, while observed cycles increase from 7.2% to 14.8% (Table 2). The diagnostic distinction directly motivates our intervention's joint stopping rule and structural edit guard.

Resource identity carries the localization gain. The four-arm comparison in Table 3 separates a resource's type from its logical identifier. Identity-only feedback exceeds type-only feedback by 8.5pt (clustered CI $[5.1, 12.3]$), with 81 paired scanner gains and 16 losses. Its acceptance is close to full localization, whereas type-only feedback is close to omitting the location clause. All four arms

Outcome	Without loc. %	With loc. %	Δpt	95% CI
Scanner gate	61.6	70.3	+8.8	[+5.0, +12.9]
Lint	25.6	25.6	+0.0	[−1.0, +1.0]
Joint scanner + lint	20.9	20.8	−0.1	[−1.2, +0.8]
AWS validation	80.3	72.9	−7.3	[−11.0, −3.9]
Cycles (E3004)	7.2	14.8	+7.6	[+3.8, +11.6]

Table 2: **Feedback localization improves scanner acceptance without improving joint validity.** Haiku; 765 matched diagnostic cells; compilation disabled; at most three new revision calls; scanner stopping. Intervals resample requirements with their five draws together. AWS validation is read-only template acceptance, not resource creation.

retain the same rule, severity, message, and finding order. The ablation changes the location clause only; resource names already embedded in messages remain shared across arms.

Feedback arm	Location information	Scanner passes	Scanner %
No location	Clause omitted	471/765	61.6
Type only	Resource type	475/765	62.1
Identity only	Logical resource identifier	540/765	70.6
Full location	Resource/property location	538/765	70.3

Table 3: **Identity, rather than type alone, explains the location contrast.** The diagnostic study: 153 requirements, five shared draws, compilation disabled, $B = 3$ additional calls, and scanner stopping. These are scanner-only acceptance rates; Table 2 gives the structural validity outcomes.

The gain concentrates on bucket obligations. On 203 bucket-bearing first shots, locations add 33.5pt scanner acceptance (CI $[23.2, 44.4]$); on 562 bucket-free shots the difference is -0.2pt (CI $[-2.3, 1.9]$). Of the bucket-bearing shots, 196 initially trigger the access-logging rule. In contrast, the bucket-free stratum with an open log-group KMS rule (`CKV_AWS_158`) has a -1.2pt location contrast (CI $[-5.8, 3.1]$). The relevant distinction is the resource obligation's shape, not simply whether a rule mentions another resource.

Rule encoding explains a concrete failure route. The logging rule applies to S3 buckets. Creating another S3 bucket to satisfy it exposes the new bucket to the same rule; choosing an intrinsic self-reference then creates a cycle. The log-group encryption rule instead points to a KMS key, which is not itself a log group. A registry census identifies 23 presence checks among 133 rules and 16 reference-shaped candidates. Schema/type adjudication labels one as same-type: `CKV_AWS_18`. These observations identify a construction target: a distinct logging resource with an explicit external boundary, applied under the guard described in §3.2. App. E records the stratum definitions and census procedure.

6 Intervention: Compile Before Refinement

6.1 Higher joint acceptance with fewer calls

Our method achieves the highest joint acceptance among the six tested procedures while using the fewest new model calls (Table 4). With Haiku 4.5 and $B = 6$, joint acceptance reaches 88.2%, compared with Reflexion's 80.4% and direct feedback's 66.7%. The paired gains are 7.8pt (CI $[1.3, 14.4]$) and 21.6pt (CI $[13.7, 29.4]$), respectively. Scanner and lint acceptance are 90.2% and 94.8%.

The quality gain accompanies lower runtime consumption: 1.78 new calls per task, versus 3.42 for Reflexion and 2.97 for direct feedback. Mean input consumption is 120.4k tokens, compared with Reflexion's 230.1k and direct feedback's 199.0k. Section 6.4 shows how this tradeoff changes with the call cap.

6.2 Compilation and execution

Execution adds value beyond library context. Executing the compiled library raises joint acceptance by 4.6pt (CI $[1.3, 8.6]$) and saves 0.52 calls per task (CI $[0.31, 0.72]$) relative to supplying the

Method	Scanner % ↑	Lint % ↑	Joint pass % ↑	New calls ↓
Self-Refine	57.5	46.4	37.9 [30.6, 45.8]	4.61
Reflexion	85.6	92.2	80.4 [73.4, 85.9]	3.42
Self-Debugging	72.5	94.1	69.9 [62.3, 76.6]	3.48†
Claude Code	45.8	69.3	40.5 [33.1, 48.4]	5.46
Direct feedback	68.0	**95.4**	66.7 [58.9, 73.6]	2.97
Ours	**90.2**	94.8	**88.2** [82.2, 92.4]	**1.78**

Table 4: **Method comparison at a common model-call budget.** Haiku 4.5; 153 identical initial candidates; at most six new calls; joint scanner-and-lint stopping for all methods. Brackets give 95% Wilson intervals for joint acceptance. Rates retain all 153 tasks. Calls exclude shared initial generation and offline compilation. †One Self-Debugging outcome has unknown quality after a provider error. Its mean-call bound is 3.477–3.484; failed-request tokens and billing remain unknown.

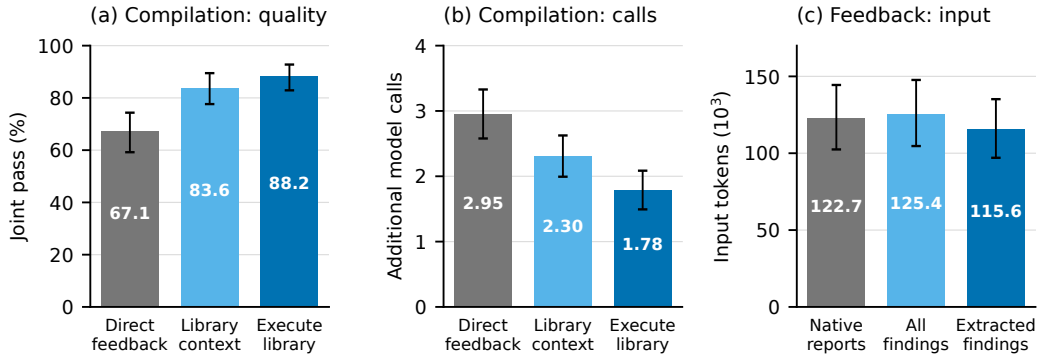


Figure 3: **Compilation strategies and feedback representations.** Haiku 4.5; $B = 6$; joint scanner-and-lint stopping. (a,b) compare raw rules, library context without execution, and guarded execution on 152 commonly scored tasks. (c) fixes execution and compares native reports, normalized all-findings reports, and extracted findings on 151 commonly scored tasks. Calls and input tokens exclude shared initial generation and offline compilation. Bars show means with pointwise 95% paired-bootstrap intervals. One context-only lint execution error and two native-report context-limit rejections remain unknown in the full 153-task results.

library as context. The comparison fixes requirements, schemas, authored contracts, revision prompt, and acceptance criterion. On the 152 commonly scored tasks in Figure 3, direct feedback, library context, and guarded execution reach 67.1%, 83.6%, and 88.2% joint acceptance. Table 7 reports all 153 tasks, including unknowns.

Table 5 separates two contributions on that same paired cohort. Library context improves acceptance and reduces consumption; executing the library adds a further gain while saving 35.6k input tokens per task. This comparison isolates execution more directly than the method comparison, whose baselines also differ in loop strategy. The intervals quantify the joint quality–consumption improvement without equating an equal call cap with equal cost.

Paired change	Δ Joint pass (pt)	Δ New calls	Δ Input tokens (k)
Context − direct	+16.4 [+9.2, +24.3]	−0.645 [−0.954, −0.362]	−41.2 [−62.5, −21.4]
Execution − context	+4.6 [+1.3, +8.6]	−0.520 [−0.724, −0.309]	−35.6 [−49.8, −21.2]
Execution − direct	+21.1 [+13.2, +28.9]	−1.164 [−1.520, −0.809]	−76.9 [−101.5, −52.6]

Table 5: **Both library context and guarded execution improve the tradeoff.** Haiku; 152 commonly scored tasks; $B = 6$; joint stopping. Brackets are 95% paired requirement-bootstrap intervals. Negative call/token changes are savings. Token counts exclude shared initial generation and offline compilation. Full scheduled outcomes remain in Table 7.

Construction resolves failures before model revision. On the identical 153 initial drafts, guarded construction alone raises joint acceptance from 6 to 22 passes: 16 gains and no losses, with zero model calls. Scanner passes rise from 16 to 82, while lint passes remain 43. Construction handles the

supported rule obligations; subsequent model revisions address the remaining scanner and structural failures.

6.3 Feedback representation

With compilation and rechecking fixed, native feedback supplies Checkov JSON and IAM findings; all-findings feedback normalizes rule, severity, message, and location; extraction additionally selects blocking findings and removes exact duplicates. On the 151 commonly scored tasks, extracted-minus-native joint acceptance is $+2.6$pt (CI $[-2.0, 7.3]$), and input-token change is -7.1k (CI $[-22.7, 7.9]$). These intervals include zero, so the experiment does not identify an incremental benefit from extraction. The all-findings comparison leads to the same conclusion (Table 8). Native reports exceed the context limit on two tasks; their outcomes and missing usage remain explicit in App. C.

6.4 Budget sensitivity and token consumption

Our method reaches Reflexion's six-call acceptance rate with a two-call cap: at $B = 2$, Ours achieves 80.4% joint acceptance (Figure 4a). Increasing our cap to six raises acceptance to 88.2%; Claude Code rises from 6.5% to 40.5% over those caps. Auxiliary critique/reflection calls consume the cap.

The token view shows the same quality–consumption benefit over Reflexion (Figure 4b,c). At $B = 6$, Ours uses 120.4k input and 5.55k output tokens per task for 88.2% joint acceptance, versus Reflexion's 230.1k/6.19k and 80.4%. Claude Code uses 430.2k input but only 3.49k output tokens for 40.5% acceptance. Each point pairs joint acceptance with the input or output tokens consumed by that procedure.

6.5 Runtime-model transfer

Direct feedback already reaches near-ceiling acceptance with the stronger runtimes. Opus 5.5 confirms $152/153$ joint passes for Ours (99.3%; one unknown) versus $153/153$ for direct feedback. Fable 5.1 confirms $147/153$ (96.1%; four unknown) versus $150/153$ (98.0%). Both models refine the same Haiku-generated drafts. The commonly scored pairs have identical joint verdicts, so the Haiku acceptance advantage does not persist in this comparison. Tables 9 and 10 report all methods and token consumption; App. D.1 gives bounds for unknown outcomes.

7 Scope and Limitations

Generalization. The 153 requirements were used during development and have one selected initial draw each. The diagnostic study adds repeated draws on the same requirements, not new tasks. Bootstrap intervals describe cohort variation rather than development-selection bias. A held-out test requires freezing the library, guards, prompts, and stopping rule before evaluating new tasks. Runtime transfer reuses Haiku drafts; fresh generation by each model is another open comparison.

Attribution and acceptance. The Ours–Reflexion contrast measures complete procedures. The fixed-prompt ablation isolates execution relative to library context, but a Reflexion-plus-library control and individual guard ablations were not run. The matched intervention measures the effect of compilation and execution, rather than mediation of the location effect. Re-encoding the logging rule and rerunning the location ablation would test that causal mechanism directly. Compilation relies on authored mappings for six constructor families. Joint acceptance checks a fixed rule pack and structural validity, including a minimum resource-count guard; it does not establish full requirement satisfaction, broader security, or successful deployment.

Cost scope. Token results measure runtime consumption under a call cap. Offline compilation and human authoring time are unmeasured, and token-budget-matched comparisons remain open. Early termination and missing usage in model transfer limit unconditional cost comparisons.

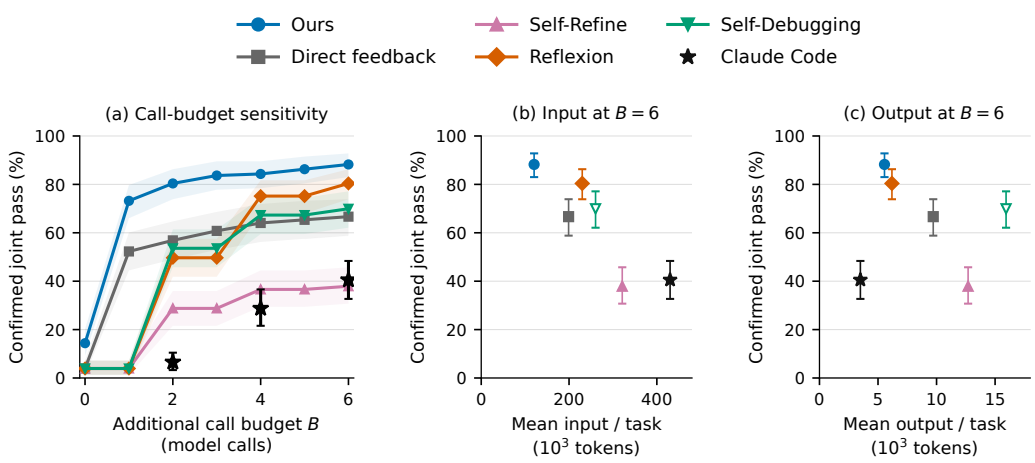


Figure 4: **Budget sensitivity and measured token consumption.** Haiku 4.5; the same 153 initial drafts as Table 4. (a) Additional model-call cap $B = 0, \ldots, 6$: completed loop-round prefixes and separate Claude Code runs at each cap; our $B = 0$ point includes compiled edits. (b,c) Joint acceptance versus mean recorded input/output tokens at $B = 6$. All costs exclude shared initial generation; input includes cache creation and reads. Bands and vertical bars show pointwise 95% requirement-bootstrap intervals for confirmed joint pass. One Self-Debugging outcome is unknown at $B \geq 4$; its hollow token markers show recorded-cost lower bounds because that request's usage is missing. All denominators remain 153.

8 Conclusion

Rule diagnostics reveal resource obligations that a scanner score alone can obscure. Compiling supported obligations and checking their execution turns that diagnosis into a useful intervention: higher joint acceptance with fewer calls and tokens on the evaluated Haiku tasks. The fixed-prompt ablation shows that executing the library adds value beyond supplying it as context. New tasks with a frozen pipeline are the next generalization test.

Reproducibility Statement

Supporting materials include the prompts, feedback renderers, shared initial candidates, refinement outputs, verifier implementations, and analysis code needed to reproduce the reported comparisons. Model identifiers, decoding settings, tool versions, sampling protocols, and statistical procedures are specified. Tables and independent static evaluations can be recomputed from these outputs without new generation. Repeating generation requires access to the stated model endpoints.

Ethics Statement

The study uses public benchmark requirements without personal data. Generated templates are evaluated with static tools. The reported acceptance criterion concerns rule-pack compliance and structural validity; it is not a deployment or security certification.

LLM Usage Disclosure

Language models are experimental subjects throughout, as specified in §4. The authors also used LLM-based coding and writing assistants to analyze experimental outputs, draft analysis code, and edit prose. Authors verified the quantitative claims and take full responsibility for the manuscript. No LLM generated any reported ground-truth verdict or served as the outcome judge.

References

Lakshya A. Agrawal, Aditya Kanade, Navin Goyal, Shuvendu K. Lahiri, and Sriram K. Rajamani. Guiding language models of code with global context using monitors. *arXiv preprint*

arXiv:2306.10763, 2023.

Benjamin Agyekum and Fabio Santos. Does fixing break security? an empirical study of security degradation in iterative LLM-driven infrastructure-as-code repair. *arXiv preprint arXiv:2608.13404*, 2026.

Manar Alsaid, Chimdumebi Nebolisa, and Faris Abbas. TerraProbe: A layered-oracle framework for detecting deceptive fixes in LLM-assisted Terraform. *arXiv preprint arXiv:2606.26590*, 2026.

Amazon Web Services. Supported foundation models in Amazon Bedrock. https://docs.aws.amazon.com/bedrock/latest/userguide/models-supported.html, 2026. Experiment-specific requested identifiers are reported in the protocol.

Anthropic. Claude Code: Overview. https://code.claude.com/docs/en/overview, 2026. Official documentation, accessed September 25, 2026.

Maria Camporese and Fabio Massacci. Repairing vulnerabilities without invisible hands. a differentiated replication study on LLMs. *arXiv preprint arXiv:2507.20977*, 2025.

Lokesh Chauhan. IaC-Guard-V: A verification framework for LLM-generated infrastructure-as-code repairs. *arXiv preprint arXiv:2609.28488*, 2026.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. In *International Conference on Learning Representations (ICLR)*, 2024.

Dekun Dai, MingWei Liu, Anji Li, Jialun Cao, Yanlin Wang, Chong Wang, Xin Peng, and Zibin Zheng. FeedbackEval: A benchmark for evaluating large language models in feedback-driven code repair tasks. *arXiv preprint arXiv:2504.06939*, 2025.

Jonas Gehring, Kunhao Zheng, Jade Copet, et al. RLEF: Grounding code llms in execution feedback with reinforcement learning. *arXiv preprint arXiv:2410.02089*, 2024.

Prithwish Jana, Sam Davidson, Bhavana Bhasker, Andrey Kan, Anoop Deoras, and Laurent Callot. TerraFormer: Automated infrastructure-as-code with LLMs fine-tuned via policy-guided verifier feedback. *Proceedings of the International Conference on Software Engineering (ICSE)*, 2026. arXiv:2601.08734.

Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy. InferFix: End-to-end program repair with LLMs. *arXiv preprint arXiv:2303.07263*, 2023.

Mohamed Jouini. Verifier-first evaluation of agentic LLMs for infrastructure-as-code generation. *arXiv preprint arXiv:2607.20478*, 2026.

William Franz Lamberti. Line-anchored feedback cuts token costs and improves correctness in AI code editing. *arXiv preprint arXiv:2607.12713*, 2026.

Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. TBar: Revisiting template-based automated program repair. *arXiv preprint arXiv:1903.08409*, 2019.

Aman Madaan, Niket Tandon, Prakhar Gupta, et al. Self-refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

Minase Mekete Mengistu, Juri Di Rocco, Phuong T. Nguyen, and Davide Di Ruscio. TerraRepair: A tool-grounded LLM agent for infrastructure-as-code repair. *arXiv preprint arXiv:2607.11390*, 2026.

Roman Nekrasov, Stefano Fossati, Indika Kumara, Damian Andrew Tamburri, and Willem-Jan van den Heuvel. IaC generation with LLMs: An error taxonomy and a study on configuration knowledge injection. *arXiv preprint arXiv:2512.14792*, 2025.

Theo X. Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. Is self-repair a silver bullet for code generation? *arXiv preprint arXiv:2306.09896*, 2023.

Mayur Amarnath Palavalli and Mark Santolucito. Using a feedback loop for LLM-based infrastructure as code generation. *arXiv preprint arXiv:2411.19043*, 2024.

Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. Asleep at the keyboard? assessing the security of GitHub Copilot's code contributions. In *IEEE Symposium on Security and Privacy (S&P)*, 2022.

Neil Perry, Megha Srivastava, Deepak Kumar, and Dan Boneh. Do users write more insecure code with AI assistants? In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2023.

Francesco Romeo, Luigi Arena, Francesco Blefari, Francesco Aurelio Pironti, Matteo Lupinacci, and Angelo Furfaro. ARPaCCino: An agentic-RAG for policy as code compliance. *arXiv preprint arXiv:2507.10584*, 2025.

Nuno Saavedra, João F. Ferreira, and Alexandra Mendes. InfraFix: Technology-agnostic repair of infrastructure as code. *arXiv preprint arXiv:2503.17220*, 2025.

Torsten Scholak, Nathan Schucher, and Dzmitry Bahdanau. PICARD: Parsing incrementally for constrained auto-regressive decoding from language models. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2021.

Melika Sepidband, Hung Viet Pham, and Hadi Hemmati. On the role of fault localization context for LLM-based program repair. *arXiv preprint arXiv:2604.05481*, 2026.

Changhee Shin, Bom Kim, and Seungsoo Lee. ALPS: Automated least-privilege enforcement for securing serverless functions. *arXiv preprint arXiv:2603.25393*, 2026.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

Brandon T. Willard and Rémi Louf. Efficient guided generation for large language models. *arXiv preprint arXiv:2307.09702*, 2023.

W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, 2016.

Yinsheng Yao, Hongxiang Zhang, Weixi Tong, and Tianyi Zhang. FLARE: Fine-grained diagnostic feedback for LLM code refinement. *arXiv preprint arXiv:2606.03852*, 2026.

He Ye and Martin Monperrus. ITER: Iterative neural repair for multi-location patches. *arXiv preprint arXiv:2304.12015*, 2023.

Tianyi Zhang, Shidong Pan, Zejun Zhang, Zhenchang Xing, and Xiaoyu Sun. Deployability-centric infrastructure-as-code generation: Fail, learn, refine, and succeed through LLM-empowered DevOps simulation. *Proceedings of the ACM on Software Engineering (FSE)*, 2026. arXiv:2506.05623; GitHub: Tianyi2/IaCGen, Apache-2.0.

A PROTOCOL, ACCOUNTING, AND REPRODUCIBILITY

The intervention experiments use 153 CloudFormation requirements and one shared Haiku draft per requirement. The compiler-disabled diagnostic study uses five shared draws per requirement, as specified in App. E. The call cap B counts additional successful model responses. The component study is specified in App. C.

Common-call-budget comparison. Table 4 uses the prospectively selected first draw for all 153 requirements. There are 917 scored method–task pairs and one Self-Debugging provider error; its quality remains unknown. Before compilation or refinement, $6/153$ candidates pass both scanner and lint. Self-Refine retains prior candidates and self-critiques; Reflexion supplies the latest failed candidate, findings, and reflection to revision; Self-Debugging retains candidate/explanation/feedback history and introduces the current findings after explanation. These are source-grounded IaC adaptations, not reproductions of the original benchmark protocols. Baselines receive raw rule/schema context; our method uses compiled context and deterministic construction. Claude Code 2.1.281 uses the same candidates, model, raw reference context, and joint gate, with a hard six-call request proxy. It chooses its own file/shell operations and stopping sequence; the harness scores its final artifact. Its 40.5% joint acceptance therefore describes this constrained configuration, not unrestricted agent capability. Auxiliary agent calls consume the cap.

The Self-Refine adaptation shares the 153 initial candidates and permits at most three complete critique/refinement rounds within the six-call cap. It does not receive checker findings, but the harness checks scanner and lint initially and after each revision and stops on joint acceptance. It uses 706 successful critique/refinement calls, or 4.61 per scheduled task, excluding the shared initial generation, with no unknown final outcomes. A round contains two model calls, so equal refinement caps would not imply equal call budgets.

A.1 OUTCOME DEFINITIONS AND DENOMINATORS

Unknown outcomes. An unknown outcome lacks a reliable final quality judgment because of a provider, verifier or process failure. It is distinct from an observed parse, lint or scanner failure. With k confirmed passes and u unknowns among n scheduled tasks, the reported confirmed rate is k/n and the pass-rate identification interval is $[k/n, (k+u)/n]$. A Wilson or bootstrap sampling interval describes a different source of uncertainty and does not resolve missing outcomes. For example, Self-Debugging has 107 confirmed joint passes and one unknown among 153: its identification interval is 69.9–70.6%. This one missing outcome cannot change the method ranking in Table 4; our method and Reflexion have no unknowns there. Paired quality comparisons use cases with known outcomes in both methods and report their smaller denominator; such comparisons are conditional on obtaining both judgments. When missingness differs across methods, these conditional comparisons can be selective; we do not assume missing outcomes would have the same pass rate as observed ones. Missing provider usage or rejected-request billing is not imputed as zero.

Scanner predicates. For CloudFormation, the gate requires a parseable template, no HIGH/MEDIUM findings after the fixed advisory reconciliation, and the benchmark resource guard. The IAM rule permits documented actions that require wildcard resources; it does not treat every wildcard as a violation. The guard requires at least two resources when multiple resources are expected, and otherwise at least one. Checker execution failure is not a clean result.

IAM effect audit. On all 917 available final artifacts from Table 4, removing false-positive wildcard-resource findings on explicit `Deny` statements removes 149 findings from 13 method–task outcomes across all six methods. No scanner or joint acceptance verdict changes; each affected artifact retains other blocking findings. The one provider-error outcome remains unknown. This final-artifact audit leaves recorded Checkov/lint judgments fixed and cannot undo earlier feedback, policy edits, or consumed calls.

A.2 BUDGET AND COST ACCOUNTING

Figure 4 uses the joint-stopping comparison. Its five loop policies have budget-independent prompts, so we evaluate completed-round prefixes at $B = 0, \ldots, 6$. Accepted artifacts carry forward; an

auxiliary stage without its revision is not a new candidate. Claude Code receives the actual call cap in its prompt and is therefore run separately at $B = 2, 4, 6$, with 153 outcomes at each cap. The two extra agent comparisons consume 294 and 586 successful model calls. They have no unknown outcomes; the original Self-Debugging provider error remains unknown at $B \geq 4$. Shared initial generation and offline compilation are excluded from B.

Observed token and request costs. The same $B = 6$ trajectories also support token accounting, which is distinct from imposing an equal token cap. The following means include every successful new response per scheduled task; input includes uncached input, cache creation and cache reads without double-counting nested usage categories. Token columns use thousands of tokens. Request

Method	Input (10^3 tokens)	Output (10^3 tokens)	Request time (s)
Direct feedback	199.0	9.72	42.8
Self-Refine	321.2	12.70	73.8
Reflexion	230.1	6.19	32.4
Self-Debugging[†]	260.4	15.94	96.6
Claude Code	430.2	3.49	26.3
Ours	120.4	5.55	25.2

Table 6: **Runtime tokens and request time on the main Haiku cohort.** Means per scheduled task at $B = 6$; shared initial generation and offline compilation are excluded.

time sums recorded request/response elapsed time; it excludes controller, tool and deployment waits and is not hardware-matched end-to-end runtime. [†]Self-Debugging's unaccounted provider attempt makes its recorded costs lower bounds. Input and output have different prices, and cached input can also be priced differently. Neither equal call caps nor equal total-token counts imply equal dollar cost. Figure 4a retains the controlled unit, model calls; panels b,c show these token expenditures against joint acceptance at $B = 6$.

A.3 Statistical analysis

The main comparison uses 20,000 paired requirement resamples and Wilson intervals for marginal rates. Component effects use 10,000 paired resamples on 152 commonly scored tasks for compilation and 151 for feedback. All-task tables retain their unknowns. Budget curves use 5,000 resamples and pointwise intervals. Runtime transfer combines 10,000 paired resamples with unknown-outcome identification bounds and reports Holm-adjusted tests across the 33 planned contrasts. Diagnostic intervals use 10,000 requirement resamples, retaining all five draws together. All pairing is by requirement; no independence is assumed between methods on one requirement.

A.4 Frozen inputs and prompt construction

The shared initial drafts, requirements, raw rules, schemas, authored mappings and compiled library are frozen in the experiment source package. The primary runner is `analysis/joint_quality_v2_20260926/loop_run.py`; its `common_prompt` function combines the requirement, full candidate, feedback, and method context. The baseline stage prompts are in `analysis/loop_baselines/policies.py`. The component runner is `analysis/joint_ablation_20260926/run.py`, using the frozen prompt and feedback builders from `analysis/compilation_ablation/run.py`. These executable builders and per-request prompt hashes specify the evaluated prompts.

The native-report condition returns Checkov JSON and IAM findings; all-findings feedback retains normalized rule, severity, message, and location; extracted feedback selects blocking findings and deduplicates them. Parser and lint diagnostics are supplied in each condition. The guard, rule pack, final-artifact selection, and stopping predicate remain fixed.

A.5 Model providers and recorded identifiers

The experiments invoke Anthropic Claude models through Amazon Bedrock (Amazon Web Services, 2026). The requested inference-profile identifiers below are recorded by the runners; a model alias without a dated suffix is not an immutable weight-version identifier. Runtime-model transfer holds the Haiku initial drafts fixed.

Paper label	Requested Bedrock identifier
Haiku 4.5	`us.anthropic.claude-haiku-4-5-20251001-v1:0`
Opus 5.5	`us.anthropic.claude-opus-5-5`
Fable 5.1	`us.anthropic.claude-fable-5-1`

Haiku supplies the primary generation/refinement runs. Opus and Fable supply the runtime-transfer study. The primary and transfer runners record request/response identities and do not substitute another model after a restriction. Their protocols and usage receipts are included with the corresponding experiment artifacts.

B COMPILER AND RUNTIME DETAILS

Compilation prepares reusable rules; construction applies them to a candidate without generator calls. The matched experiment uses a frozen metadata-driven, hand-assisted library, whose exact inputs and identity are specified in §C. Its contracts and guarded execution are specified below.

B.1 INPUTS, AUTHORED MAPPINGS, AND EXECUTION

The compiler reads the scanner registry's inspected paths, expected values and check kinds, including the `ANY_VALUE` presence marker. Resource schemas provide required properties, types and enumerations. In the evaluated registry, CloudFormation has 133 checks: 81 value checks, 7 negative checks and 45 custom-code checks, with 23 `ANY_VALUE` keys. Custom code can omit the metadata needed to construct a repair.

An authored nine-row property-name lexicon maps reference-shaped paths to witness types. Its six supported families are KMS keys, S3 logging buckets, SQS dead-letter queues, log groups for access-log destinations, WebACLs, and Glue security configurations. The remaining mappings identify security groups, cluster subnet groups and IAM roles, which this walk refuses as network or authorization intent. The intent partition is also authored. Finding a reference shape does not authorize constructing its target or changing an author-set value.

For a recognized `ANY_VALUE` path, compilation records an obligation edge. A constructor starts from the target type's schema skeleton; bounded verification fills supported failing value checks and completes dependencies until stable or refused. Schema-required siblings of touched objects are completed as well: Kinesis `StreamEncryption`, for example, needs a `KeyId` binding. Reference vocabulary selects `Ref` versus `GetAtt Arn`, with `cfn-lint` checking the resulting shape. These checks cover schema constraints, not every provider requirement.

Derived and authored contributions. The metadata-driven CloudFormation compiler derives 19 edges and adds five by hand for `CKV_AWS_116`, 195, 119, 149 and 173, whose relevant keys are absent from metadata. Against a 14-edge authored reference table, derivation recovers 11 and adds ten edges the table lacked. The six constructor types above contain 1, 6, 1 and 2 derived checks for KMS, S3, SQS and log groups, respectively; WebACL and Glue instead require two and one hand patches, including WebACL's schema one-of choice.

Runtime and rechecking. The executor discovers applicable obligations, constructs and binds supported finite bundles, and refuses protected or unsupported edits. It serializes CloudFormation without YAML aliases and rechecks the candidate before forming model feedback. Extraction reconciles scanner and permission findings, retains rule, severity, message and resource/property location, selects blocking residuals and removes exact duplicates. The generator receives the requirement, revised candidate and residual report; its next output is constructed and rechecked again. Acceptance requires the joint scanner-and-lint predicate, with at most B additional calls. The exact prompt builders are listed in App. A.4.

B.2 OBLIGATION SEMANTICS

Definition 1 (Obligation graph)**.** For resource types T in rule pack C, the obligation graph $\mathcal{G}_C$ is directed on T, with an edge $t \to t'$ whenever discharging a rule on type t requires a resource of type t' to exist. A witness is the resource that discharges an obligation.

Under LETTER semantics, $\mathcal{G}_C^{\ell}$ records only what the checker literally requires. Under DENOTATIONAL semantics, $\mathcal{G}_C^{d}$ additionally requires the named witness to exist with the required type. Platform admissibility is a separate predicate. A presence check on the bearer can therefore accept a value without inspecting its referent.

For `CKV_AWS_18`, `LoggingConfiguration` presence suffices. A symbolic `!Ref AppBucket` on `AppBucket` passes that test but creates a CloudFormation dependency cycle; a literal self-name need not create the same dependency edge. Repeatedly creating fresh, initially unsatisfied logging buckets would instead produce a cascade. The construction bundle creates a separate `LogBucket`, binds its logging destination to a parameter naming a distinct existing bucket, and points `AppBucket` at `LogBucket`. The caller supplies the external destination and permissions. This ends the supported chain without proving that the boundary exists or that the stack deploys. The runtime checks for structural regressions on the candidate being edited.

B.3 CONSTRUCTION CONTRACTS AND PROOF

Type-level ordering alone cannot establish the following contracts.

- **Finite admitted bundles.** Each supported obligation has a finite precompiled bundle. Bundle dependencies have a finite component order; each cyclic component is handled as one finite bundle or refused. Separately satisfiable resource types do not suffice to establish this condition.
- **Protected paths.** Contracts identify author-set and intent paths. Admitted bundles leave them unchanged and refuse unsupported obligations. This syntactic condition does not prove preservation of all natural-language intent.
- **Preservation.** Insertion and rebinding preserve every previously satisfied checker predicate and introduce no blocking finding on new declarations. Admission and rechecking must establish this on applicable inputs; locality and ordering alone do not imply it.
- **Reference closure.** Internal bindings resolve to existing or newly created declarations. External bindings are explicit boundary inputs whose existence and permissions remain caller obligations, rather than fabricated literals.

Proposition 2 (Conditional construction guarantees)**.** Assume the four contracts above. Process a finite set of applicable obligations in dependency order, applying each admitted bundle at most once and refusing any step that violates its contract. The pass terminates with zero runtime generator calls, preserves previously satisfied checker predicates, introduces no unresolved internal reference, and leaves protected paths unchanged. Graph ordering costs $O(|T| + |E|)$; total cost also includes finding discovery, constructor execution, emitted output and verification.

Proof. Finite bundles give finitely many steps with finite work per step; unresolved cyclic components are bundled or refused, not recursively expanded. The deterministic executor makes no generator call. Induct on committed steps: the preservation contract retains every previously true predicate and prevents new blocking findings on inserted declarations; refusal leaves the artifact unchanged. Reference closure resolves introduced internal references, and the protected-path contract retains author values. These properties therefore compose. A topological order of the component DAG takes linear graph time, which does not bound the cost of emitted declarations or their verification. □

Preservation is substantive: even resource-local predicates $p = q$ and $p = 1$ show how setting a flagged p can break an already satisfied equality. Uniqueness, cardinality, negative cross-resource constraints, quotas and template-size limits can also defeat insertion. Finite regression checks are evidence about an implementation, not a proof over all templates. Internal closure does not establish external permissions, platform acceptance or deployment; the compiler and execution pass in this experiment use no model calls.

B.4 MEASURED CALL ACCOUNTING AND REUSE

For direct feedback and Ours, cumulative joint-accepted counts before additional call $b = 1, \ldots, 6$ are $(6, 80, 87, 93, 98, 100)$ and $(22, 112, 123, 128, 129, 132)$, respectively. Neither procedure has an early terminal non-acceptance before the cap in this cohort. Summing the unaccepted tasks at each step therefore gives

$$N_{\text{direct}} = \sum_{b=1}^{6}(153 - A_{\text{direct},b-1}) = 454, \qquad N_{\text{ours}} = \sum_{b=1}^{6}(153 - A_{\text{ours},b-1}) = 272.$$

These totals agree with the response receipts. They account for 182 fewer runtime calls; token consumption is measured separately in Table 6. The library is reused across tasks, but compilation and human-authoring effort were not timed. Consequently, the experiment reports runtime savings without estimating a total-cost break-even number of tasks.

C MATCHED COMPILATION AND RESIDUAL-FEEDBACK ABLATIONS

Shared inputs and fixed procedure. The component study uses the same 153 Haiku-generated initial candidates and six-call budget as Table 4. Haiku 4.5 supplies all runtime calls. Requirements, scanner/schema context, authored contracts, revision prompt, output limit, parser, structural diagnostics, and joint acceptance remain fixed. The metadata-derived library includes disclosed authored mappings and patches; this library is held fixed across the context and execution conditions. The frozen input is `replication/compilation_ablation_20260924/inputs/compiled_library.json`; the executor uses `analysis/joint_quality_v2_20260926/compiler_adapter.py`. The same library and guarded compiler support Table 4. "Guarded execution + extracted findings" is the Ours condition in that table. Direct feedback is the same raw-rule control named in Table 4; the model interprets its rules. Library context supplies the compiled constructions without executing them. Guarded execution applies them before refinement and after each candidate, preserving property types, intrinsic expressions, and policy containers and rejecting new structural diagnostics. Every condition stops on joint scanner-and-lint acceptance.

With execution fixed, native feedback supplies original Checkov JSON and IAM findings. All-findings feedback normalizes rule, severity, message, and location. Extraction additionally selects blocking findings and removes exact duplicates. These comparisons separate executable construction from report representation, and report representation from field selection. Library execution makes no model calls.

Configuration	Scanner %	Lint %	Joint %	New calls	Unknown
Direct feedback	68.0	95.4	66.7	2.97	0
Library context only	87.6	90.8	83.0	2.30	1
Guarded execution + extracted findings	90.2	94.8	88.2	1.78	0
Guarded execution + native reports	89.5	92.8	85.0	1.69	2
Guarded execution + all findings	90.2	93.5	85.0	1.88	0

Table 7: **Component ablations with all 153 tasks retained.** Haiku 4.5; at most six additional model calls; joint stopping. Unknowns are unconfirmed outcomes, not established code failures. Calls count successful new responses, excluding shared initial generation and compilation; rejected-request billing is unknown.

Execution failures and denominator. The context-only condition has one internal cfn-lint error while processing a conditional resource expression. Native reports exceed the 200,000-token context limit twice (227,929 and 214,986 reported tokens). No shortened replacement prompt is substituted. These three outcomes retain their original condition and remain in the all-task denominator. All other conditions have 153 known final verdicts. The 1,625 successful new calls across the five conditions have provider usage receipts; rejected-request token usage and billing are unavailable. The direct-feedback and guarded-execution controls are the same outcomes used in the primary comparison, not additional independent replications.

Paired effects. Figure 3 and Table 8 use 152 commonly scored tasks for the three compiler conditions and 151 for the three feedback conditions. All conditions in a panel retain the same

requirements. Intervals use 10,000 paired requirement resamples; they are pointwise 95% intervals. The all-task outcomes above retain the excluded unknowns. No cache creation/read tokens are reported for successful calls in this component study, so provider input tokens equal total input context.

Comparison	n	**Δjoint (pt)**	**Δnew calls**	**Δinput (10^3)**
Context − direct	152	$+16.4$ $[+9.2, +24.3]$	-0.645 $[-0.954, -0.362]$	-41.2 $[-62.5, -21.4]$
Execution − context	152	$+4.6$ $[+1.3, +8.6]$	-0.520 $[-0.724, -0.309]$	-35.6 $[-49.8, -21.2]$
Execution − direct	152	$+21.1$ $[+13.2, +28.9]$	-1.164 $[-1.520, -0.809]$	-76.9 $[-101.5, -52.6]$
Extracted − native	151	$+2.6$ $[-2.0, +7.3]$	$+0.026$ $[-0.179, +0.238]$	-7.1 $[-22.7, +7.9]$
Extracted − all	151	$+3.3$ $[-0.7, +7.9]$	-0.113 $[-0.272, +0.046]$	-9.8 $[-21.8, +1.8]$

Table 8: **Paired component effects on commonly scored tasks.** Every difference is the first named condition minus the second. Higher joint acceptance and lower calls/input tokens are preferred. Brackets give pointwise 95% paired-bootstrap intervals. Extraction's quality and cost intervals include zero for both comparisons.

Execution improves joint acceptance over library context and reduces model calls. The extraction point estimates favor compact feedback, but neither its joint-quality nor its token difference from native reports is statistically resolved. Context-limit failures also prevent an unconditional token-cost comparison based only on successful receipts. The guarded zero-call construction effect and remaining structural errors are reported in §6.2 and App. D.1.

D Runtime-model transfer and structural outcomes

D.1 Scanner compliance and structural validity

Tables 9 and 10 apply the metrics in §4.4 to the guarded, jointly stopped runs. Scanner and lint judge the same final artifact. Parse failures do not pass lint; verifier/provider execution errors remain unknown and stay in the planned denominator. The Haiku rows use exactly the primary-comparison records.

Runtime-model transfer. All three runtime models refine the same Haiku-generated drafts. For Opus 5.5, Ours confirms $152/153$ joint passes (99.3%), with one unknown, versus $153/153$ for direct feedback. For Fable 5.1, the corresponding counts are $147/153$ (96.1%), with four unknowns, and $150/153$ (98.0%). The joint-pass difference allowed by unknown outcomes is $[-0.65, 0.00]$pt for Opus and $[-1.96, +0.65]$pt for Fable. There are no discordant joint verdicts among the 152 and 149 commonly scored pairs, respectively. These comparisons do not establish a joint-acceptance advantage for compilation on either runtime model. Recorded new calls are 133 versus 160 for Opus and 155 versus 191 for Fable; early execution failures and missing usage prevent treating these totals as an unconditional cost advantage.

Across the 2,754 scheduled model–method outcomes, 44 have unknown quality: one for Haiku, 29 for Opus and 14 for Fable. In particular, 25 Opus Self-Refine trajectories ended under the campaign stop before their next model request; they are not observed quality failures. Started trajectories retain their original outcomes. The transfer analysis uses 10,000 paired requirement resamples over all 153 IDs, with an interval envelope spanning the differences allowed by unknown outcomes. For Ours minus direct feedback, these 95% envelopes are $[-1.96, 0.00]$pt for Opus and $[-4.58, +1.96]$pt for Fable. Exact McNemar tests condition on known pairs; Holm adjustment covers the 33 planned method and runtime-model contrasts.

Quality and token consumption. Tables 9 and 10 report mean input and output tokens separately, in thousands per scheduled task. Input includes cache creation and reads; provider-reported nested token categories are not counted twice. For Haiku, Ours uses 120.4k input and 5.55k output tokens, versus 199.0k and 9.72k for direct feedback, while raising joint acceptance from 66.7% to 88.2%. The corresponding observed token reductions are 39.5% and 42.9%. The stronger-runtime rows report observed expenditure, including early terminations; their unknown outcomes and missing usage limit quality–cost comparisons. Token counts avoid dependence on changing prices, but equal token counts across models or token categories do not imply equal monetary cost.

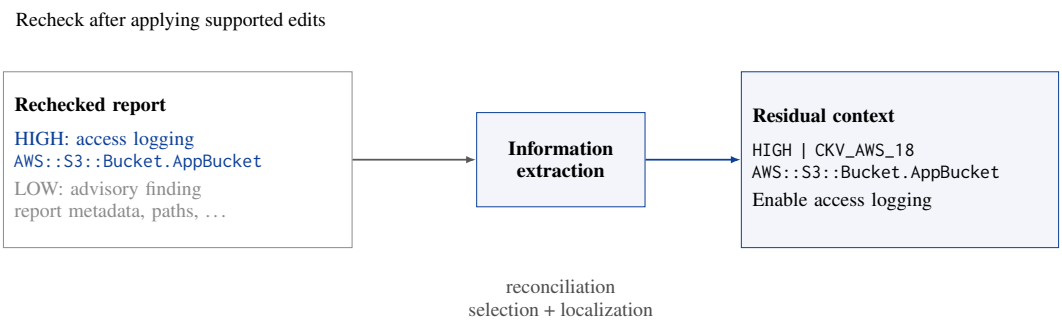


Figure 5: **Information extraction preserves the residual diagnosis and its target.** The schematic shows a blocking finding retained with its resource location, while advisory content and report overhead are omitted. The model also receives the requirement, revised candidate, and applicable failure history. This diagram describes the representation; its incremental quality and token benefits over native reports are unresolved.

Method / configuration	Scanner %	Lint %	Joint %	Unknown	New calls	Input (10^3)	Output (10^3)
Haiku 4.5							
Direct feedback	68.0	95.4	66.7	0	2.97	199.0	9.72
Self-Refine	57.5	46.4	37.9	0	4.61	321.2	12.70
Reflexion	85.6	92.2	80.4	0	3.42	230.1	6.19
Self-Debugging	72.5	94.1	69.9	1	3.48	260.4†	15.94†
Claude Code	45.8	69.3	40.5	0	5.46	430.2	3.49
Ours	90.2	94.8	88.2	0	1.78	120.4	5.55
Opus 5.5							
Direct feedback	100.0	100.0	100.0	0	1.05	103.9	6.11
Self-Refine	74.5	83.0	74.5	25	2.41	245.1	11.92
Reflexion	100.0	100.0	100.0	0	2.03	200.5	6.53
Self-Debugging	100.0	100.0	100.0	0	2.01	204.8	13.03
Claude Code	95.4	98.0	95.4	3	2.65	286.9	5.10
Ours	99.3	99.3	99.3	1	0.87	88.8	5.29
Fable 5.1							
Direct feedback	98.0	98.7	98.0	0	1.25	122.1	9.56
Self-Refine	83.7	98.7	83.7	1	2.80	285.2	18.48
Reflexion	99.3	99.3	99.3	1	2.01	199.0	6.46
Self-Debugging	98.7	99.3	98.7	1	2.07	209.4†	12.88†
Claude Code	92.8	95.4	92.8	7	3.44	390.0†	7.73†
Ours	96.1	96.7	96.1	4	1.01	100.8†	7.38†

Table 9: **Joint static outcomes of the guarded method and baselines.** Each row uses the same 153 Haiku-generated initial drafts, one per requirement, with at most six additional model calls and joint scanner-and-lint stopping. The guarded compiler preserves property types, intrinsic expressions and policy containers. Unknown outcomes remain in the 153-task denominator. Model labels identify the runtime model. Costs are means per scheduled task, excluding shared initial generation and offline compilation. Input includes uncached, cache-creation and cache-read tokens; output includes provider-reported thinking tokens without double counting. †Recorded token lower bound: unknown-request usage is missing. Costs include early-terminated trajectories and do not establish a matched-quality or monetary-cost advantage.

The zero-call comparison covers all 153 shared first shots. Before construction, 16 pass the scanner, 43 pass lint and 6 pass both. After guarded construction, the counts are 82, 43 and 22, respectively. All 16 joint-pass gains and zero losses are on identical initial candidates. This is the initial construction effect, distinct from final results that also include model refinement.

E Diagnostic feedback and rule-shape evidence

The diagnostic evidence uses the Codex manuscript's recorded compiler-disabled feedback study. The only changed feedback field is the resource/property location; both arms retain rule, severity and message. A finding's message may itself name a resource. Each arm uses the same five initial draws

Method / configuration	Scanner %	Lint %	Joint %	Unknown	New calls	Input (10^3)	Output (10^3)
Haiku 4.5							
Direct feedback	68.0	95.4	66.7	0	2.97	199.0	9.72
Ours	90.2	94.8	88.2	0	1.78	120.4	5.55
Opus 5.5							
Direct feedback	100.0	100.0	100.0	0	1.05	103.9	6.11
Ours	99.3	99.3	99.3	1	0.87	88.8	5.29
Fable 5.1							
Direct feedback	98.0	98.7	98.0	0	1.25	122.1	9.56
Ours	96.1	96.7	96.1	4	1.01	$100.8^{\dagger}$	$7.38^{\dagger}$

Table 10: **Structural outcomes with direct feedback and guarded compiled execution.** Each row uses the same 153 Haiku-generated initial drafts, one per requirement, with at most six additional model calls and joint scanner-and-lint stopping. The guarded compiler preserves property types, intrinsic expressions and policy containers. Unknown outcomes remain in the 153-task denominator. Model labels identify the runtime model. Costs are means per scheduled task, excluding shared initial generation and offline compilation. Input includes uncached, cache-creation and cache-read tokens; output includes provider-reported thinking tokens without double counting. †Recorded token lower bound: unknown-request usage is missing. Costs include early-terminated trajectories and do not establish a matched-quality or monetary-cost advantage. The comparison changes compiled context and its execution together; it does not isolate execution alone. The context-only control appears in Table 7.

per requirement, and one model call performs one revision. Thus the three-round diagnostic cap is also $B = 3$ additional model calls. The final artifact is the first scanner-accepted output or the last output within the cap. Paired intervals resample 153 requirements with all their draws together.

E.1 Outcomes and strata

Table 2 is reconstructed from `docs/evidence/rebuttal-metric-validity/per_artifact_external_validity.jsonl`; its stored clustered estimates are in `analysis.json` in that directory. Lint uses cfn-lint 1.55.1 on the recorded template text. AWS validation uses the read-only `ValidateTemplate` operation in `us-east-1`. Scanner, lint, and their conjunction are recorded for the same artifact.

Strata use bucket presence in the shared initial draft, before either feedback treatment. The source `docs/evidence/bucket-strata/analysis.json` retains cell-level pass indicators and rule exposure. Requirement counts overlap across strata when bucket presence varies across draws. Two initial templates were replaced in the source cache; their stratum labels and unparseable inputs are flagged, with sensitivity analyses retained in that source.

Initial-draft stratum	Cells	Δgate pt	95% CI	Gained:lost
Bucket-bearing	203	$+33.5$	$[+23.2, +44.4]$	$69:1$
Bucket-free	562	-0.2	$[-2.3, +1.9]$	$20:21$
Bucket-free, CKV_AWS_158 open	162	-1.2	$[-5.8, +3.1]$	$5:7$

Table 11: **With-location minus without-location scanner acceptance.** The log-group KMS row is a subset of the bucket-free stratum. All intervals cluster by requirement.

E.2 What the census determines

The source `docs/evidence/obligation-graph/selfloop_class_census.json` enumerates 133 unique Checkov CloudFormation rules over 77 resource types. Registry metadata mechanically identifies 23 any-string-satisfiable presence checks; property-name shapes select 16 possible reference obligations. The schema/type verdicts for those candidates are authored and documented: one same-type and 15 cross-type. All candidates are adjudicated. The same-type case is the S3 access-logging rule. This is a reproducible syntactic census plus type adjudication, not a general decision procedure for arbitrary custom checker code.

A same-type obligation can be satisfied by naming the bearer under a presence-and-type semantics: self-witnessing is available, not unique. CloudFormation additionally interprets intrinsic references

as dependency edges, so an intrinsic self-reference fails its cycle check. Our construction instead uses a distinct logging bucket and an explicit parameter for a further external destination. The parameter closes the supported construction chain; its existence and permissions remain caller obligations.

The bucket concentration and cycle diagnostics motivate this mechanism. The guarded intervention evaluates joint acceptance and runtime cost, not the location contrast after compilation. A checker-reencoding experiment has not been run. The old construction-first scanner rates and location-attenuation estimates are not used as intervention evidence here.