INTERACTIVE CLARIFICATION FOR CLOUD INFRASTRUCTURE-AS-CODE SYNTHESIS

Anonymous authors
Paper under double-blind review

ABSTRACT

The scale and complexity of modern cloud infrastructure have made "Infrastructure-as-Code" (IaC) essential for managing deployments through declarative configurations. While large language models (LLMs) are increasingly used to generate IaC configurations from natural language, user requests are often ambiguous and underspecified. Unlike traditional code generation, it is costly and time-consuming to test IaC configurations during synthesis, forcing the LLMs into an almost one-shot regime. We observe that ambiguity in IaC synthesis exhibits a compositional structure: configurations decompose into three axes (resources, topology, attributes) where higher-level decisions constrain lower-level ones. We propose a training-free, multi-level disambiguation framework that generates diverse candidate specifications, identifies structural disagreements across these axes, ranks them by informativeness, and produces targeted clarification questions that progressively narrow the configuration space. We further introduce AMBIG-IAC, a benchmark of 300 validated IaC tasks with expert-verified ambiguous requests, and define evaluation metrics based on graph edit distance and exact attribute matching. Comprehensive experiments show that our method achieves the best structural fidelity at every interaction budget and the best attribute fidelity at larger budgets, at a fraction of the cost of information-gain alternatives, with gains that scale with the interaction budget and generalize across models. Extensive ablation studies and analyses further demonstrate its robustness for interactive IaC synthesis.

1 INTRODUCTION

LLMs are increasingly adopted for synthesizing structured artifacts from natural-language descriptions: code, database queries, and configuration programs (Ding et al., 2025; Zhou et al., 2025; Mu et al., 2024). However, user descriptions that drive this synthesis are often underspecified—prompts state high-level goals, not implementation details, so many valid outputs can satisfy the same prompt. A model that silently "resolves" this ambiguity via its own priors may produce plausible but misaligned results. Cloud Infrastructure-as-Code (IaC) is a particularly demanding instance of this problem (Yang et al., 2025a). IaC frameworks such as Terraform (HashiCorp, 2025b) define cloud infrastructure as declarative configurations (Yang et al., 2025a), and recent work has begun generating them directly from natural language (Kon et al., 2024). A single user request typically specifies only high-level infrastructure goals, leaving lower-level decisions (e.g., compute abstractions, resource connectivity, and security or networking policies) underspecified. This ambiguity gives rise to a large space of plausible yet potentially misaligned configurations (Figure 1).

One way to handle ambiguity is by iterative clarification and refinement. In conventional code generation, agents refine programs using execution feedback: running tests, observing failures, and repairing. Cloud configuration offers no such loop. Executing an IaC program requires provisioning real infrastructure, which is slow, costly, and often irreversible or subject to strict rate limits. Dry-run validators such as `terraform plan` catch syntactic and type errors but cannot detect semantic misalignment with user intent. Indeed, we show that roughly 90% of generated configurations pass Terraform validation regardless of intent alignment (Section 6.6). Cloud agents, therefore, operate in an almost one-shot regime where incorrect decisions incur real cost and are difficult to undo (Grigorev, 2026). Hence, precise disambiguation is a first-class requirement in IaC generation.

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.

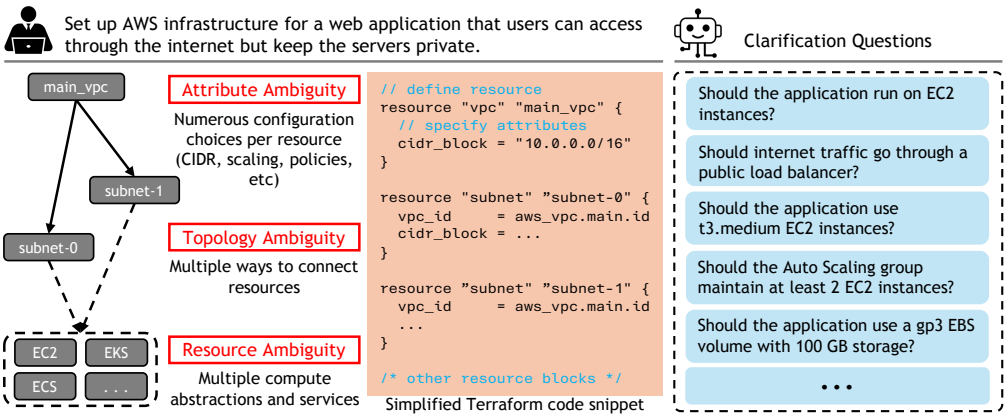


Figure 1: An underspecified user request can be satisfied by many plausible cloud infrastructure configurations, each represented as a resource dependency graph (left). Ambiguity arises in choosing resource abstractions and services for compute, networking, storage, in inter-resource topology, and in per-resource configuration attributes, when translating intent into Infrastructure-as-Code (right).

Existing clarification methods do not transfer readily to cloud IaC because they target different ambiguity spaces. First, prior work primarily studies open-domain questions that rely on common-sense reasoning and admit a small set of salient interpretations (Min et al., 2020; Andukuri et al., 2024; Chen et al., 2025a). In such settings, clarification questions are relatively straightforward to formulate, and ambiguity can often be resolved within two to three turns (Zhang et al., 2025; Andukuri et al., 2024; Mu et al., 2024). IaC instead exhibits compositional ambiguity across resources, topology, dependencies, and attributes, producing a vast space of plausible configurations. Second, methods from code generation rely on *execution feedback*. They identify ambiguity through behavioral divergence among candidate programs on synthesized tests (Mu et al., 2024; Lahiri et al., 2023). This is difficult for IaC because execution may provision or modify real cloud resources, so ambiguity must be resolved before deployment. Third, existing approaches provide little guidance on where to direct clarification effort. Recent work confirms the consequence empirically: without structural guidance, LLMs waste their question budget on low-value clarifications (Vijayvargiya et al., 2025). As Figure 1 illustrates, IaC synthesis explores a structured configuration space where resources, topology, and attributes are tightly coupled, which we leverage to guide the clarification process.

Our central idea is to use the explicit structure of candidate artifacts to guide clarification. IaC configurations decompose into three axes (*resources*: which components to provision; *topology*: how they connect; *attributes*: what settings to apply), which expose disagreements that can be computed symbolically and used to construct targeted questions, reducing the need to evaluate freely generated questions against candidate solutions. We combine this mechanism with cross-axis shortlisting and preservation of surviving candidates, and evaluate whether it improves configuration fidelity under a fixed interaction budget. The framework generates diverse candidate specifications, identifies structural disagreements across the three axes, ranks them by informativeness, and uses the most informative disagreements to generate targeted clarification questions. After the user answers each question, incompatible candidates are pruned, and the process repeats, progressively narrowing the configuration space.

This paper makes the following contributions:

- We present the first investigation of interactive cloud IaC synthesis that exploits the structure of IaC configurations to guide multi-turn user intent clarification.
- We propose a training-free clarification framework that derives questions from symbolic disagreements across resources, topology, and attributes, balances the disagreements presented to the question generator, and preserves surviving candidates through the final output.
- We introduce AMBIG-IAC, a benchmark of 300 IaC tasks with validated reference configurations and expert-verified ambiguous prompts. We additionally propose an evaluation framework that measures configuration correctness along structure (via graph edit distance) and concrete resource–attribute values (via exact, occurrence-aware matching).

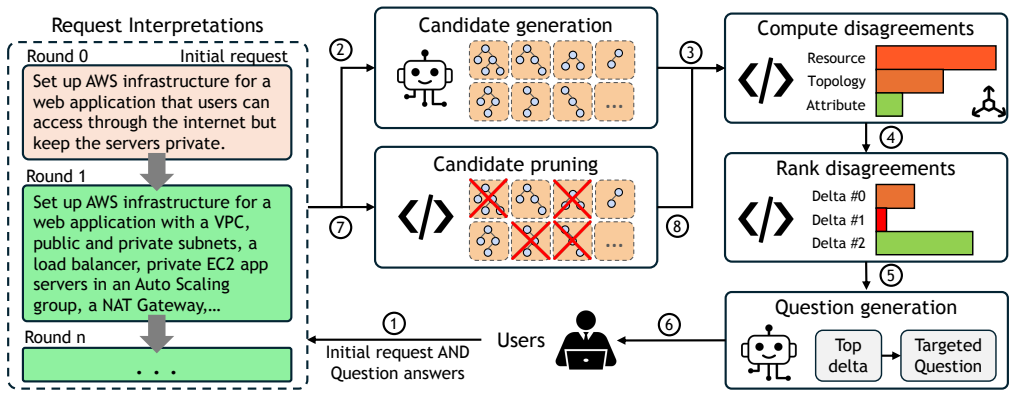


Figure 2: Overview of the iterative disambiguation process for interactive Infrastructure-as-Code synthesis. 1. Users provide an initial IaC request. 2. Initially, a pool of diverse structured specifications is generated based on the current interpretations. 3. Symbolically compute disagreements among the current pool of candidates. 4. These disagreements are ranked to identify the most informative differences measured by entropy. 5. Given the shortlisted disagreements, the system generates a targeted clarification question. 6. Interaction with the user. 7. Prune candidates inconsistent with the user's answer. 8. The refined candidate pool is fed back into the next iteration, repeating the process until the interaction budget is exhausted; pool regeneration is triggered.

- Our evaluation shows that our method outperforms baselines on structural fidelity at every interaction budget, with gains that grow with the budget, and generalizes across language models, at a fraction of the computational cost of information-gain alternatives. Our ablation studies further demonstrate its robustness.

2 Motivation

IaC frameworks such as Terraform make cloud infrastructure reproducible and auditable through declarative, version-controlled configurations (Amazon Web Services, 2026; HashiCorp, 2025a; Yang et al., 2025b), but authoring these configurations requires substantial knowledge of cloud services and provider-specific semantics (Srivatsa et al., 2024). This has fueled growing interest in using LLMs to generate IaC from natural language, as reflected in recent surveys and benchmarks studying cloud infrastructure generation and editing (Srivatsa et al., 2024; Kon et al., 2024). However, generating correct IaC configurations is challenging, because it requires mapping underspecified user requests to complex, interdependent cloud configurations (Kon et al., 2024). The operational stakes are real: in one reported incident, an AI coding agent executed a Terraform command that deleted the production infrastructure, including a database holding 2.5 years of course submissions and its automated backups, requiring roughly 24 hours of recovery (Grigorev, 2026).

3 Problem Formulation

We study the problem of generating IaC configurations from ambiguous natural-language requests through interactive clarification. A user's request expresses their *explicit intent* I: the initial, underspecified prompt. Because I leaves many decisions open, it admits multiple plausible interpretations, which in turn induce a large space $\mathcal{C}(I)$ of distinct candidate configurations. We assume that each request corresponds to a latent, intended target, represented in our setting by a reference configuration $c^* \in \mathcal{C}(I)$ that satisfies not only the explicit intent I but also the user's *implicit intent*, the unstated constraints and preferences. Since multiple syntactic realizations may represent the same design (e.g., different naming conventions), we normalize configurations to remove naming and representational differences. Architecturally distinct configurations, however, correspond to different interpretations and are therefore not treated as interchangeable. The agent cannot observe the implicit intent, and thus cannot identify c^* directly; it can only elicit information about it by asking the user a bounded number of clarification questions, progressively narrowing $\mathcal{C}(I)$ toward c^*.

Structured specifications. Ambiguity in IaC synthesis arises along three axes: **resources**, **topology**, and **attributes**. Together, these axes define the combinatorial space of infrastructure configurations. IaC configurations consist of a set of resource blocks (see Figure 1), and we represent each configuration as a structured triple $s = (R, T, A)$, where R is a multiset of resource types, $T \subseteq R \times R$ is a set of directed dependency edges (typed by their source and target resource types), and A assigns to each resource a set of configuration key/value pairs. This decomposition induces a structured interface that supports more efficient and systematic intent elicitation by LLM-based agents.

Interactive objective. The agent is given a fixed budget of K binary clarification questions. At each round $t \in \{1, \ldots, K\}$, it selects a question q_t, receives the user's answer $a_t \in \{\mathsf{yes}, \mathsf{no}\}$, and appends (q_t, a_t) to the interaction history H_t. After the budget is exhausted, the agent returns a single specification $\hat{c}$. The goal is to make the final configuration as close as possible to the target c^*. Our evaluation measures this discrepancy using structural graph edit distance and exact resource-attribute matching.

4 METHOD

Since c^* is unobserved and $\mathcal{C}(I)$ is intractably large, we approximate the configuration space with a finite *candidate pool* $\mathcal{P}$ and use disagreement among candidates as a proxy for where the agent's uncertainty about c^* concentrates. We model LLMs' alternative interpretations as a set of structured specifications, analyze the axes along which they disagree, and construct clarification questions based on the most informative disagreement. The agent (Figure 2) iteratively asks one question per round, prunes the pool based on the user's answer, and regenerates candidates when the pool is exhausted. After the budget is spent, a surviving candidate is returned.

Candidate generation. The candidate pool $\mathcal{P} = \{s_1, \ldots, s_N\}$ serves as a concrete approximation of the space of plausible configurations $\mathcal{C}(I)$. Given the user's request (and, in subsequent rounds, the accumulated question/answer history H_t), the agent generates N candidate specifications via temperature sampling, so that they explore different resource selections, topology wiring, and attribute choices. Candidates with identical resource types and topology edges are deduplicated to prioritize architectural diversity; attribute disagreements are computed among the retained candidates, so attribute-only alternatives within an otherwise identical structure may be removed.

Computing disagreements. Because each candidate is a structured triple (R, T, A), disagreements can be computed symbolically along each axis. For resources, the agent extracts the set of resource types from each candidate and records any type that is present in some candidates but absent from others. For topology, it extracts the set of directed dependency edges and similarly records edges on which candidates disagree. For attributes, it groups candidates by resource type, then for each shared attribute key, checks whether candidates assign different values. Each disagreement partitions the pool into the candidates on either side of the difference.

Ranking disagreements. Not all disagreements are equally informative. A disagreement that has a vast majority carries little discriminative value, whereas an even split maximizes the information gained from a single binary question. We therefore prioritize disagreements with diverse candidate support, measured by the Shannon entropy of the induced partition: resource and topology disagreements induce present/absent partitions, while attribute disagreements induce partitions over distinct values (including *unset*), so attribute entropy measures value diversity rather than the information gain of the eventual binary question. Disagreements below an entropy floor are filtered out, with the unfiltered set retained if none passes. Within each axis, disagreements are ranked by entropy, and the per-axis rankings are interleaved round-robin into a shortlist of at most five entries; this balances the evidence presented to the question generator without enforcing a fixed sequence of question axes.

Question generation and pruning. In each round, the LLM receives the shortlisted disagreements and candidate summaries, and generates one natural-language yes/no question (e.g., whether the infrastructure should include a NAT gateway) that a user can answer without knowledge of the underlying candidates, together with a candidate *keep set* for each answer. Given the user's answer a_t, the agent deterministically retains the candidates in the corresponding keep set and discards the rest: one answer can eliminate multiple candidates at once, efficiently narrowing the pool. This avoids an additional candidate-evaluation call, although pruning correctness depends on the generated question and keep sets faithfully expressing the underlying disagreement.

Table 1: Human validation of AMBIG-IAC and IaC-Eval on 100 samples, with normalized ratings and inter-coder agreement measured by Gwet's AC. Higher ratings indicate greater naturalness, ambiguity, and target consistency.

Dataset	Natural.	Ambig.	Target Cons.
Normalized score (0–1)			
IaC-Eval	0.25	0.24	0.90
AMBIG-IAC	**0.78**	**0.89**	**0.99**
Inter-coder agreement (Gwet's AC)			
IaC-Eval	0.71	0.82	0.70
AMBIG-IAC	0.65	0.82	0.98

Regeneration and termination. Regeneration is triggered when the pool is exhausted (fewer than two candidates remain). The regeneration procedure repeats candidate construction, conditioning on both the original request and the full accumulated interaction history. This lets the model produce candidates consistent with the information gathered so far while still exploring the remaining uncertainty; the structured representation makes newly generated candidates easy to deduplicate and filter against prior interactions, avoiding revisiting resolved disagreements. The interaction terminates when the budget K is exhausted or when the agent can no longer produce structurally distinct candidates, indicating that its uncertainty has been sufficiently resolved. Upon termination, the agent returns a surviving candidate; if none survives, it performs one final generation conditioned on the original request and the full interaction history.

5 EXPERIMENTAL SETUP

Dataset. We construct AMBIG-IAC, a benchmark of 300 Terraform tasks sourced from IaC-Eval (Kon et al., 2024). We manually verified and fixed the reference programs using `terraform plan`, filtering out programs that could not be validated. The dataset covers 167 unique AWS resource types across 44 service families (e.g., VPC, S3, IAM, RDS, Lambda). To introduce realistic ambiguity, we use an LLM to rewrite each original task prompt into a high-level, goal-oriented request that preserves consistency with the reference configuration but admits multiple plausible architectures. For example, "Configure a weighted routing policy that splits users between two db_instances that are replicas of a main db_instance" becomes "Our database is getting slow for users in other regions—need to fix that and make reads faster globally." A manual inspection of 100 sampled instances (performed by three domain experts) confirms that the rewritten prompts are natural yet answerable tasks: relative to IaC-Eval, AMBIG-IAC scores substantially higher on Naturalness and Ambiguity while preserving Target consistency, with substantial-to-almost-perfect inter-coder agreement (Table 1).

User Proxy. We simulate user responses with an LLM-based oracle, following prior work (Zhou et al., 2025; Mu et al., 2024; Vijayvargiya et al., 2025; Kobalczyk et al., 2025; Zhang et al., 2025). Given a ground-truth Terraform configuration and a binary clarification question, the proxy answers based solely on whether the reference configuration implies yes or no, and is explicitly instructed to answer strictly from the reference, with no external knowledge or assumptions. All methods interact with the same oracle under the same question budget; this standardizes the simulated-user model and interaction budget across methods.

Baselines. Under the same question budget K, we compare against four interactive baselines:

- **Direct Question Generation**: the LLM directly generates clarification questions.
- **Best-of-N**: at each round the LLM generates N candidate questions conditioned on prior Q&A, and a separate ranker LLM selects one.
- **Self-Consistency**: N candidate questions are generated, embedded, and clustered; the question closest to the centroid of the largest cluster is selected, favoring questions that the model consistently considers important.
- **Active Task Disambiguation (ATD)** (Kobalczyk et al., 2025): ranks candidate questions by expected information gain, estimated over a grid of candidate questions $\times$ solutions; evaluated at $K \in \{5, 10\}$ due to high costs.

Table 2: Main results on AMBIG-IAC using GPT-4o-mini. We report means with 95% task-level bootstrap CIs over 300 tasks (10,000 resamples). Struct. is normalized-GED similarity; R+A F1 is exact-match resource-and-attribute F1; and Δ gives each baseline's absolute and relative difference from ours. ATD is evaluated at $K \in \{5, 10\}$ due to its substantially higher cost.

Method	Budget K	Struct. (%)	Δ Struct.	R+A F1 (%)	Δ F1
Direct Q Generation	5	43.08 [39.90, 46.31]	-5.87 (↓11.99%)	17.69 [15.40, 20.05]	-4.35 (↓19.74%)
Best-of-N	5	41.28 [37.97, 44.68]	-7.67 (↓15.67%)	18.18 [15.80, 20.68]	-3.86 (↓17.51%)
Self-Consistency	5	38.59 [35.48, 41.73]	-10.36 (↓21.16%)	15.63 [13.40, 17.97]	-6.41 (↓29.08%)
ATD	5	42.23 [38.55, 45.85]	-6.72 (↓13.73%)	**22.70** [20.21, 25.21]	+0.66 (↑2.99%)
Ours	5	**48.95** [45.95, 51.91]	–	22.04 [19.57, 24.63]	–
Direct Q Generation	10	44.38 [41.31, 47.52]	-6.60 (↓12.95%)	18.72 [16.58, 20.97]	-5.49 (↓22.68%)
Best-of-N	10	43.52 [40.24, 46.79]	-7.46 (↓14.63%)	18.91 [16.48, 21.34]	-5.30 (↓21.89%)
Self-Consistency	10	38.41 [35.27, 41.71]	-12.57 (↓24.66%)	16.30 [14.04, 18.66]	-7.91 (↓32.67%)
ATD	10	41.89 [37.95, 45.83]	-9.09 (↓17.83%)	24.01 [21.38, 26.69]	-0.20 (↓0.83%)
Ours	10	**50.98** [48.02, 53.98]	–	**24.21** [21.62, 26.87]	–
Direct Q Generation	15	46.31 [43.28, 49.42]	-8.53 (↓15.55%)	18.45 [16.33, 20.62]	-8.08 (↓30.46%)
Best-of-N	15	43.12 [39.86, 46.48]	-11.72 (↓21.37%)	19.52 [17.08, 22.04]	-7.01 (↓26.42%)
Self-Consistency	15	38.19 [34.95, 41.39]	-16.65 (↓30.36%)	16.12 [13.81, 18.50]	-10.41 (↓39.24%)
Ours	15	**54.84** [51.87, 57.86]	–	**26.53** [23.88, 29.28]	–

All baselines share the same final generation step: after K rounds of Q&A, the accumulated interaction history is used to construct a clarified intent, from which the final specification is generated.

Metrics. We evaluate generated specifications against the ground-truth reference along the two axes of IaC ambiguity: whether the infrastructure has the right *structure*, and whether its concrete *values* are exactly right. Both metrics operate on specs normalized to a common label namespace. **Structure** (Struct.) scores graph similarity via the Graph Edit Distance (GED) between the reference and generated resource graphs, using unit costs for node/edge insertions, deletions, and cross-type substitutions. It jointly penalizes missing or extra resources *and* incorrect wiring; the score is $\max(0, 1 - \mathrm{GED}/D)$, where D is the total node and edge count of both graphs, and same-type nodes are interchangeable (Appendix C.3), so 1.0 is an exact structural match. **Res+Attr F1** (R+A F1) is a deliberately strict, exact-match test of concrete configuration values. Each spec is reduced to an occurrence-aware multiset of resource instances and attribute key–value pairs (multiplicity counts: three `aws_subnet` in the reference but two generated yields at most two matches), and the score is the exact-match F1 between the two multisets after canonicalizing booleans, numbers, and letter case. We grant no embedding or fuzzy credit, because values that are close in form can be incompatible in behavior (`10.0.0.0/16` vs. `10.1.0.0/16`); free-form naming keys and cross-resource references (already scored by *Structure*) are excluded. Both metrics measure recovery of the latent reference configuration rather than satisfaction of the initial underspecified request alone; they can therefore penalize valid alternative realizations. In our audit of 76 low-Structure outputs, an equivalent Terraform encoding was the primary cause of structural mismatch in one case (1.3%; Section 6.5 and Appendix I); this audit does not quantify alternative-realization penalties across all outputs or attribute values (Appendix C.5).

6 RESULTS AND ANALYSIS

Our evaluation centers on four research questions:

- **RQ1:** Does structure-aware disambiguation outperform structure-agnostic baselines?
- **RQ2:** Does structure-aware disambiguation performance scale with the number of rounds?
- **RQ3:** Does the method generalize across different LLMs?
- **RQ4:** Are the reported gains statistically robust?

We then analyze where the method fails and verify deployment-time validity; ablations of hyperparameters and cross-axes balancing are in Appendices F and G.

Table 3: Average gain per 5 extra questions (GPT-4o-mini), in points and relative (%).

Method	Δ Struct.	Δ R+A F1
Direct Q Gen.	+1.62 (↑3.68%)	+0.38 (↑2.19%)
Best-of-N	+0.92 (↑2.25%)	+0.67 (↑3.62%)
Self-Consistency	-0.20 (↓0.52%)	+0.25 (↑1.59%)
Ours	**+2.95 (↑5.86%)**	**+2.25 (↑9.71%)**

Table 4: Seeded runs (Claude Opus 4.6, K=5): mean ± std over three seeds.

Method	Struct.	R+A F1
Direct Q Generation	53.46 ±0.37	25.75 ±0.60
Best-of-N	57.91 ±0.24	30.37 ±0.83
Self-Consistency	57.46 ±1.25	28.83 ±0.51
Ours	**63.95** ±0.68	**31.95** ±0.32

6.1 RQ1: Overall Performance

Structure-aware disambiguation outperforms all structure-agnostic baselines: across budgets of K=5, 10, and 15, our method achieves the best Structure score at every budget and the best R+A F1 at the two larger budgets; its lead over the strongest baseline widens as the budget grows (Table 2). We observe that test-time inference techniques (Best-of-N, Self-Consistency) do not offer a meaningful improvement over direct question generation, and Self-Consistency is the weakest method overall: question consensus does not align with informativeness in a highly structured configuration space, where useful questions are those that target the specific points on which interpretations diverge. ATD, also an information-gain method, generates candidate questions and scores them over a question × solution grid rather than deriving them from computed spec deltas. It trails our Structure score while spending 4× our tokens and 11× our wall-clock per task (detailed cost results are in the supplementary material), and its one nominal edge, R+A F1 at K=5, sits well inside our confidence interval (Table 2). For controlled comparison, all methods use the same binary-question interface, which restricts the information conveyed per turn. Richer response formats and persistent user-preference models may improve absolute performance, but evaluating these extensions requires interaction data that is not currently available for IaC. The gains reflect both disagreement-guided questioning and preservation of surviving candidates. With the baselines' final-generation protocol, our question-selection strategy alone achieves 49.07 Structure and 26.01 R+A F1 at K=15, compared with 46.31 and 18.45 for direct question generation; returning a surviving candidate further raises these to 54.84 and 26.53 (Appendix E). Question selection thus improves both metrics under a shared output protocol, while candidate preservation contributes substantially to structural fidelity. ***Finding 1:*** *Combining disagreement-guided clarification with candidate preservation improves structural fidelity over the evaluated baselines, while requiring approximately one-quarter of ATD's tokens and one-eleventh of its wall-clock time at K=5.*

6.2 RQ2: Interaction Scaling

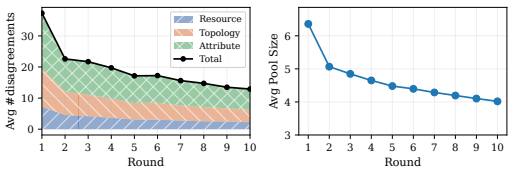
Most methods benefit from additional questions, as shown in Table 3, with ours gaining the most on both metrics. Self-Consistency even shows a slight decrease, suggesting that consensus-based selection saturates quickly. The widening gap indicates that our method makes more effective use of each additional question. Baselines generate questions without explicit awareness of which axes remain ambiguous, so later questions are more likely to be redundant or low-value. In contrast, our disagreement-driven approach continually re-estimates where ambiguity lies after each answer, directing subsequent questions to the most informative remaining differences. Figure 3a shows that disagreement counts decrease steadily across all three axes (left), and resource-level disagreements are resolved first, then topology, then attributes, while the pool size shrinks in step (right), converging toward a shared interpretation. Figure 3b examines regeneration: most tasks require 3–5 regenerations (left), indicating that the candidate pool is typically exhausted within 2–3 questions, as each question is sufficiently discriminative to eliminate a substantial fraction of candidates. We see a positive correlation between regeneration frequency and performance (right). This correlation is consistent with conditioned regeneration—producing new candidates consistent with accumulated Q&A history—acting as a mechanism for progressive refinement, though we do not isolate its causal effect. ***Finding 2:*** *Additional interaction helps the most when the agent explicitly tracks unresolved ambiguity.*

6.3 RQ3: Cross-Model Generalization

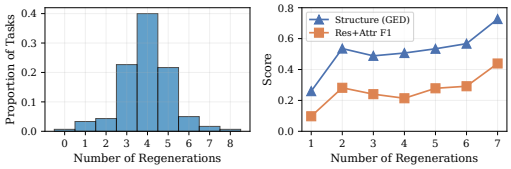
The gains generalize across backbones: we re-ran the full pipeline on five models spanning open and closed families, and our method achieves the best results on both metrics for every one of them

Table 5: Generalization across five backbone LLMs (K=5).

Model	Method	Struct.	R+A F1
Claude Opus 4.6	Direct Q Gen.	53.17	26.39
	Best-of-N	57.67	30.07
	Self-Consistency	56.95	28.87
	Ours	**64.69**	**32.31**
GPT-4.1-mini	Direct Q Gen.	50.77	22.42
	Best-of-N	53.06	23.69
	Self-Consistency	52.58	22.26
	Ours	**57.33**	**26.27**
Qwen3-Coder NEXT	Direct Q Gen.	49.05	19.62
	Best-of-N	48.17	20.28
	Self-Consistency	47.66	18.40
	Ours	**58.53**	**25.19**
Llama-4-Scout	Direct Q Gen.	48.92	17.37
	Best-of-N	50.29	18.14
	Self-Consistency	49.42	17.15
	Ours	**54.02**	**21.23**
GPT-4o-mini	Direct Q Gen.	43.08	17.69
	Best-of-N	41.28	18.18
	Self-Consistency	38.59	15.63
	Ours	**48.95**	**22.04**



(a) Per-round dynamics show the candidates converging toward a shared interpretation.



(b) Most tasks require three to five regenerations, with performance generally improving as regenerations increase.

Figure 3: Round and regeneration dynamics.

(Table 5). Stronger backbones lift all methods, but the relative ordering remains unchanged; the lead is largest on Qwen3-Coder and remains clear on the strongest backbone, Claude Opus 4.6.

6.4 RQ4: Statistical Robustness

The reported gains are statistically robust to both sampling stochasticity and task selection, which we assess in two ways. Across three seeded trials on the strongest backbone (Table 4), our lead over the strongest baseline is many times the run-to-run standard deviation on both metrics. The task-level bootstrap CIs in Table 2 tell the same story at the population level: on Structure, our interval is disjoint from every baseline's at K=10 and K=15, and essentially disjoint at K=5. Ablations of pool size and sampling temperature (Appendix F) show that the gains are insensitive to hyperparameters, while the cross-axes ablation (Appendix G) shows they depend on the structure itself: replacing round-robin shortlist construction across axes with entropy-only selection costs up to 3.19 Structure and 2.19 F1 points at K=15 (5.8% and 8.3% relative).

6.5 Failure Analysis

We inspect the interaction traces of 76 of 300 tasks on which our method scores below 0.5 on Structure with Claude Opus 4.6 (K=5) and attribute each failure to one cause (Appendix I, Table 14). Four of five failures (80.3%) originate in candidate generation rather than in the questioning loop. Most often the pool settles on the mainstream AWS service for a workload while the user intended a less common one (52.6%): ECS instead of Elastic Beanstalk for "gradual traffic shifting", ElastiCache instead of DAX in front of DynamoDB, SQS or SNS instead of an MSK cluster, Lex v2 instead of Lex v1. In other cases, every candidate wraps the requested resource in the same default scaffolding (21.1%): a VPC with an internet gateway comes back with private subnets, a NAT gateway, an EIP, and route tables in all candidates, so the pool agrees and none of it is put to the user. Both patterns reflect the prior of the generator rather than the questioning strategy, and retrieval-augmented generation over Terraform configurations is a natural complement. Within the loop, the main cause (10.5%) is compound questions such as "Do you need IAM roles and CloudWatch log groups?", which a single yes/no cannot answer; a one-resource-per-question constraint addresses this.

Within this audited subset, simulated-user errors were not identified as the primary cause of any failure, an equivalent Terraform encoding of the reference (an inline `replica` block versus `aws_dynamodb_table_replica`) accounted for one case (1.3%), and no failed specification contains a non-existent resource type. These findings suggest that candidate coverage dominates the observed structural failures, but do not establish that simulated-user errors or single-reference scoring have negligible effects across the full evaluation. The remaining failures are addressable:

they trace to the generator's coverage, which retrieval-augmented candidate generation or simply a stronger backbone can improve (Table 5), while the questioning strategy remains unchanged.

6.6 DEPLOYABILITY

Finally, we verify that generated specifications pass Terraform's own validation toolchain. We convert each specification into a program `main.tf` using an LLM, and run the standard pipeline (`init` → `validate` → `plan`, which previews changes without applying them): 90% of tasks pass on one-shot emission and 97–99% after a single error-guided repair pass, for every method. ***Finding 3:*** *Passing `terraform plan` does not guarantee alignment with user intent.*

7 RELATED WORK

Agents for cloud management. Recent work has explored LLM-based systems for cloud management, including IaC generation benchmarks (Kon et al., 2024), agentic systems for autonomous cloud operations and incident analysis (Shetty et al., 2024; Chen et al., 2025b; 2024; Yang et al., 2025b; Xiang et al., 2025; Peng et al., 2025), provider-integrated assistants such as Azure Copilot, Gemini for Google Cloud, and Amazon Q (Microsoft, 2025; Google Cloud, 2025; Amazon Web Services, 2025), and broader discussions of cloud AI agents (Yang et al., 2025a). However, they generally assume that the given tasks are well specified and do not study how an agent should resolve ambiguous user intent before acting.

Active task disambiguation. Recent work on active task disambiguation selects clarification questions by estimating how candidate answers partition a space of plausible solutions (Kobalczyk et al., 2025). Our contribution is a structure-aware implementation of disagreement-guided clarification for IaC: rather than scoring freely generated questions over a question–solution grid, we construct questions from symbolic configuration differences across resources, topology, and attributes, and retain surviving candidates through the final output. Our evaluation examines the resulting fidelity–cost tradeoff (Section 6.1), with separate ablations of cross-axis shortlisting (Appendix G) and the output protocol (Appendix E).

Clarification via fine-tuning. Prior work learns clarification behavior in bounded QA, preference-elicitation, and software-engineering settings through preference optimization, self-training, multi-turn policy learning, or persistent user modeling (Min et al., 2020; Zhang et al., 2025; Andukuri et al., 2024; Chen et al., 2025a; Zhou et al., 2025). These approaches rely on fine-tuning or learned user models. In contrast, cloud IaC is a fast-evolving domain where provider APIs and specifications change rapidly, high-quality training data is scarce, and ambiguity is structured across resources, topology, and attributes.

Clarification in code. The closest engineering analogues are code-oriented systems that resolve ambiguity by comparing competing realizations of the user's intent and querying the user on distinguishing evidence (Mu et al., 2024; Lahiri et al., 2023). Recent work in software-engineering agents further shows that interaction improves performance on underspecified tasks (Vijayvargiya et al., 2025). However, they all rely on executable artifacts, generated tests, or cheap iterative repair — feedback unavailable in IaC, where execution provisions real infrastructure and ambiguity must be resolved before deployment.

8 CONCLUSION AND FUTURE WORK

We presented a disagreement-driven disambiguation framework for interactive IaC synthesis that exploits configuration structure for multi-turn clarification, and AMBIG-IAC, a 300-task benchmark of validated IaC tasks with expert-verified ambiguous prompts. Our method outperforms structure-agnostic baselines, with gains that scale with the interaction budget and generalize across models.

Future work will explore retrieval-augmented generation to broaden candidate coverage and constraints that keep each question focused on a single decision. Evaluation with cloud practitioners will test whether gains observed with simulated users transfer to real interactions involving user uncertainty or evolving preferences.

AI USE STATEMENT

We used generative AI tools to polish the writing (grammar, wording, LaTeX formatting); all ideas, claims, experiments, and analyses are the authors' own, and every AI-assisted passage was reviewed and edited by the authors. LLMs were used to construct the dataset: an LLM rewrote the IaC-Eval prompts into ambiguous requests (Section 5), with reference programs manually verified and 100 tasks audited by three domain experts (Table 1). We did not use generative AI for research ideation, and no AI-written code was used without review. We take responsibility for the final content of this work, including text, claims, or artifacts produced with the aid of generative AI.

ETHICS STATEMENT

Human involvement was limited to three domain experts annotating 100 prompts with no personal information.

REPRODUCIBILITY STATEMENT

Dataset construction is described in Section 5; hyperparameters in Table 6; models, roles, and temperatures in Appendix B; all prompts verbatim in the Prompts section of the appendix; and both metrics in Appendices C.3 and C.4. Confidence intervals are 95% task-level bootstrap intervals (10,000 resamples over 300 tasks); run-to-run variance is in Table 4. We will open-source both the AMBIG-IAC dataset and the full code, including configurations and all interaction traces, upon acceptance.

REFERENCES

Amazon Web Services. Amazon Q in AWS services, 2025. URL https://aws.amazon.com/q/.

Amazon Web Services. What is infrastructure as code? https://aws.amazon.com/what-is/iac/, March 2026. Accessed: 2026-04-01.

Chinmaya Andukuri, Jan-Philipp Fränken, Tobias Gerstenberg, and Noah Goodman. STar-GATE: Teaching language models to ask clarifying questions. In *First Conference on Language Modeling*, 2024. URL https://openreview.net/forum?id=CrzAj0kZjR.

Maximillian Chen, Ruoxi Sun, Tomas Pfister, and Sercan O Arik. Learning to clarify: Multi-turn conversations with action-based contrastive self-training. In *The Thirteenth International Conference on Learning Representations*, 2025a. URL https://openreview.net/forum?id=SIE6VFps9x.

Yinfang Chen, Huaibing Xie, Minghua Ma, Yu Kang, Xin Gao, Liu Shi, Yunjie Cao, Xuedong Gao, Hao Fan, Ming Wen, Jun Zeng, Supriyo Ghosh, Xuchao Zhang, Chaoyun Zhang, Qingwei Lin, Saravan Rajmohan, Dongmei Zhang, and Tianyin Xu. Automatic root cause analysis via large language models for cloud incidents. In *Proceedings of the Nineteenth European Conference on Computer Systems*, EuroSys '24, 2024.

Yinfang Chen, Manish Shetty, Gagan Somashekar, Minghua Ma, Yogesh Simmhan, Jonathan Mace, Chetan Bansal, Rujia Wang, and Saravan Rajmohan. Aiopslab: A holistic framework to evaluate ai agents for enabling autonomous clouds, 2025b. URL https://arxiv.org/abs/2501.06706.

Zhongjun Ding, Yin Lin, and Tianjing Zeng. Ambisql: Interactive ambiguity detection and resolution for text-to-sql, 2025. URL https://arxiv.org/abs/2508.15276.

Google Cloud. Gemini for Google Cloud, 2025. URL https://cloud.google.com/products/gemini.

Alexey Grigorev. How I dropped our production database and now pay 10% more for AWS. https://alexeyondata.substack.com/p/how-i-dropped-our-production-database, March 2026. Substack post. Accessed: 2026-03-31.

HashiCorp. What is infrastructure as code with terraform? https://developer.hashicorp.com/terraform/tutorials/aws-get-started/infrastructure-as-code, June 2025a. Accessed: 2026-04-01.

HashiCorp. Terraform, 2025b. URL https://developer.hashicorp.com/terraform.

Kasia Kobalczyk, Nicolás Astorga, Tennison Liu, and Mihaela van der Schaar. Active task disambiguation with LLMs. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=JAMxRSXLFz.

Patrick Tser Jern Kon, Jiachen Liu, Yiming Qiu, Weijun Fan, Ting He Lei Lin, Haoran Zhang, Owen M. Park, George S. Elengikal, Yuxin Kang Ang Chen, Mosharaf Chowdhury, Myungjin Lee, and Xinyu Wang. Iac-eval: a code generation benchmark for cloud infrastructure-as-code programs. In *Proceedings of the 38th International Conference on Neural Information Processing Systems*, NIPS '24, 2024.

Shuvendu K. Lahiri, Sarah Fakhoury, Aaditya Naik, Georgios Sakkas, Saikat Chakraborty, Madanlal Musuvathi, Piali Choudhury, Curtis von Veh, Jeevana Priya Inala, Chenglong Wang, and Jianfeng Gao. Interactive code generation via test-driven user-intent formalization, 2023. URL https://arxiv.org/abs/2208.05950.

Microsoft. Microsoft copilot in Azure, 2025. URL https://azure.microsoft.com/en-us/products/copilot.

Sewon Min, Julian Michael, Hannaneh Hajishirzi, and Luke Zettlemoyer. AmbigQA: Answering ambiguous open-domain questions. In Bonnie Webber, Trevor Cohn, Yulan He, and Yang Liu (eds.), *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 5783–5797, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.emnlp-main.466. URL https://aclanthology.org/2020.emnlp-main.466/.

Fangwen Mu, Lin Shi, Song Wang, Zhuohao Yu, Binquan Zhang, ChenXue Wang, Shichao Liu, and Qing Wang. Clarifygpt: A framework for enhancing llm-based code generation via requirements clarification. *Proc. ACM Softw. Eng.*, 1(FSE), July 2024. doi: 10.1145/3660810. URL https://doi.org/10.1145/3660810.

Jingjia Peng, Yiming Qiu, Patrick Tser Jern Kon, Pinhan Zhao, Yibo Huang, Zheng Guo, Xinyu Wang, and Ang Chen. Automated lifting for cloud infrastructure-as-code programs. In *2025 IEEE/ACM International Workshop on Cloud Intelligence & AIOps (AIOps)*, 2025.

Manish Shetty, Yinfang Chen, Gagan Somashekar, Minghua Ma, Yogesh Simmhan, Xuchao Zhang, Jonathan Mace, Dax Vandevoorde, Pedro Las-Casas, Shachee Mishra Gupta, Suman Nath, Chetan Bansal, and Saravan Rajmohan. Building ai agents for autonomous clouds: Challenges and design principles. In *Proceedings of the 15th ACM Symposium on Cloud Computing*, 2024.

Kalahasti Ganesh Srivatsa, Sabyasachi Mukhopadhyay, Ganesh Katrapati, and Manish Shrivastava. A survey of using large language models for generating infrastructure as code, 2024. URL https://arxiv.org/abs/2404.00227.

Sanidhya Vijayvargiya, Xuhui Zhou, Akhila Yerukola, Maarten Sap, and Graham Neubig. Interactive agents to overcome ambiguity in software engineering, 2025. URL https://arxiv.org/abs/2502.13069.

Yiming Xiang, Zhenning Yang, Jingjia Peng, Hermann Bauer, Patrick Tser Jern Kon, Yiming Qiu, and Ang Chen. Automated bug discovery in cloud infrastructure-as-code updates with llm agents. In *2025 IEEE/ACM International Workshop on Cloud Intelligence & AIOps (AIOps)*, 2025.

Zhenning Yang, Archit Bhatnagar, Yiming Qiu, Tongyuan Miao, Patrick Tser Jern Kon, Yunming Xiao, Yibo Huang, Martin Casado, and Ang Chen. Cloud infrastructure management in the age of ai agents. *SIGOPS Operating Systems Review*, 2025a. doi: 10.1145/3759441.3759443. URL https://doi.org/10.1145/3759441.3759443.

Zhenning Yang, Hui Guan, Victor Nicolet, Brandon Paulsen, Joey Dodds, Daniel Kroening, and Ang Chen. Automated cloud infrastructure-as-code reconciliation with ai agents, 2025b. URL https://arxiv.org/abs/2510.20211.

Michael JQ Zhang, W. Bradley Knox, and Eunsol Choi. Modeling future conversation turns to teach LLMs to ask clarifying questions. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=cwuSAR7EKd.

Xuhui Zhou, Valerie Chen, Zora Zhiruo Wang, Graham Neubig, Maarten Sap, and Xingyao Wang. Tom-swe: User mental modeling for software engineering agents, 2025. URL https://arxiv.org/abs/2510.21903.

APPENDIX

This supplement contains: (i) pseudocode and sub-routine definitions for the multi-level disambiguation algorithm (Section A); (ii) the experiment configuration—models, roles, decoding settings, and method hyperparameters (Section B); (iii) formal definitions of the two evaluation metrics, the GED-based Structure score and the exact-match Res+Attr F1, together with the label normalization both share (Section C); (iv) per-task token and wall-clock costs for all methods, models, and interaction budgets (Section D); (v) an ablation of the output protocol, separating the effect of question selection from that of final-spec emission (Section E); hyperparameter and cross-axes ablations (Sections F and G); (vi) a validation of the simulated user (Section H); (vii) a failure analysis of low-scoring tasks (Section I); and (viii) the system and user prompts used at each stage of the pipeline (the *Prompts* section at the end of this supplement).

A MULTI-LEVEL DISAMBIGUATION ALGORITHM

Algorithm 1 gives the top-level loop of our method; the subsections below define each sub-routine.

A.1 NOTATION

Each candidate configuration is a structured triple

$$s = (R,\, T,\, A), \tag{1}$$

where R is a multiset of resource types, $T \subseteq R \times R$ is a set of directed dependency edges, typed by their source and target resource types, and A assigns each resource a set of configuration key–value pairs. Concretely, a resource is a label $\mapsto$ Terraform-address entry, and $\tau(a)$ denotes the type extracted from address a, e.g., $\tau(\texttt{aws_vpc.main}) = \texttt{aws_vpc}$. The agent maintains a *candidate pool* $\mathcal{P} = \{s_1, s_2, \dots\}$ approximating the space of plausible configurations, and an *interaction history* H of question–answer pairs. The agent may ask at most K binary questions; N is the pool size and θ the entropy floor.

A.2 CANDIDATE GENERATION

GENERATE(I, H, N) produces N candidates conditioned on the request I and the history H. One *anchor* spec is generated at temperature 0 under a plain generation prompt with no diversity instructions, so the pool always contains a canonical interpretation. The remaining $N-1$ specs are sampled at a higher temperature (0.7 by default).

A.3 DISAGREEMENT COMPUTATION

DISAGREEMENTS$(\mathcal{P})$ compares the candidates axis by axis; each disagreement f partitions the candidates it applies to.

- **Resources.** For each type r occurring in any candidate, let $\text{present}(r) = \{i : r \in R(s_i)\}$ and $\text{absent}(r) = \{1, \dots, |\mathcal{P}|\} \setminus \text{present}(r)$. A disagreement is recorded when both sets are non-empty.

Algorithm 1 Multi-Level Disambiguation

Input: request I; budget K; pool size N; entropy floor θ
Output: final specification $\hat{c}$

$H \leftarrow \emptyset$
$\mathcal{P} \leftarrow \textsc{Generate}(I, H, N)$
for $t = 1$ **to** K **do**
  if $|\mathcal{P}| < 2$ **then**
    $\mathcal{P} \leftarrow \mathcal{P} \cup \textsc{Generate}(I, H, N)$
  end if
  $\mathcal{D} \leftarrow \textsc{Disagreements}(\mathcal{P})$
  if $\mathcal{D} = \emptyset$ **then**
    break
  end if
  $\mathcal{D}_{>\theta} \leftarrow \{\, f \in \mathcal{D} : H_{\text{ent}}(f) > \theta \,\}$
  if $\mathcal{D}_{>\theta} \neq \emptyset$ **then**
    $\mathcal{D} \leftarrow \mathcal{D}_{>\theta}$ {drop near-consensus splits}
  end if
  $F^* \leftarrow \textsc{RoundRobinShortlist}(\mathcal{D})$ {≤ 5 diffs, interleaved across axes}
  $(q_t, \mathcal{P}^{\mathsf{yes}}, \mathcal{P}^{\mathsf{no}}) \leftarrow \textsc{Verbalize}(F^*)$
  ask q_t; observe $a_t \in \{\mathsf{yes}, \mathsf{no}\}$
  $\mathcal{P} \leftarrow \mathcal{P} \cap \mathcal{P}^{a_t}$ {deterministic prune}
  $H \leftarrow H \cup \{(q_t, a_t)\}$
end for
return $\textsc{Select}(\mathcal{P}, I, H)$

- **Topology.** The same test, per type-level directed edge: $\text{present}(e) = \{i : e \in T(s_i)\}$.
- **Attributes.** Candidates are grouped by resource type. For each type r and each key k set on r by any candidate in the group, every candidate containing r contributes a value: the value it assigns, or *unset* if it does not set k. A disagreement is recorded when these values are not all identical (values are compared after canonical JSON serialization).

A.4 Entropy Ranking and Cross-Dimension Balancing

A near-unanimous split carries little information; an even split is worth a question. Let $\{\mathcal{P}_b\}$ be the parts into which f divides the candidates it applies to—present vs. absent for resources and topology, one part per distinct value (including *unset*) for attributes. We score f by the Shannon entropy of this split,

$$H_{\text{ent}}(f) = -\sum_b \rho_b \log_2 \rho_b, \qquad \rho_b = \frac{|\mathcal{P}_b|}{\sum_{b'} |\mathcal{P}_{b'}|}. \tag{2}$$

Disagreements with $H_{\text{ent}} \leq \theta$ (0.5 in all experiments) are discarded as near-consensus splits; if the floor would discard every disagreement while the candidates still differ, the unfiltered set is used instead, so a round is never wasted. Within each axis, the survivors are ranked by descending entropy, and RoundRobinShortlist interleaves the per-axis rankings in round-robin order, keeping at most five: the shortlist F^* thus mixes the most informative disagreements from all three axes. This balancing keeps any single axis from monopolizing the question budget; the main paper's ablation shows it outperforms ranking by entropy alone.

A.5 Question Generation and Pruning

Verbalize(F^*) renders the shortlist as a structural diff summary—each disagreement listed with the candidate indices on either side—preceded by a short resource/topology synopsis of every candidate, and passes it to an LLM (the exact prompt and summary format are in the *Prompts* section). The LLM emits *one* yes/no question that targets the most informative ambiguity in the summary, phrased so a user can answer it without seeing the candidates (e.g., a resource disagreement on `aws_nat_gateway` becomes "Should the infrastructure include a NAT gateway?"); the lower-ranked shortlist entries serve as context, helping the LLM keep the question precise and non-overlapping. Together with the question, the LLM emits the keep set for either answer—the candidate indices consistent with yes and with no, read off the targeted disagreement's partition in the

Table 6: Hyperparameters of our method. N and T_{div} are swept in the main paper's sensitivity ablation; the rest are fixed across all experiments.

	Parameter	Value
K	question budget	5 / 10 / 15
N	candidates per (re)generation	10
T_{div}	diverse-sampling temperature	0.7
θ	entropy floor	0.5

summary—denoted $\mathcal{P}^{\mathsf{yes}}$ and $\mathcal{P}^{\mathsf{no}}$. Pruning therefore requires no further LLM call, although its correctness depends on the question and keep sets faithfully expressing the underlying disagreement. One answer can eliminate several candidates at once, and survivors are kept exactly as generated, so no editing errors accumulate. If the observed answer's keep set is empty or inconsistent with every surviving candidate, the pool is discarded, which triggers regeneration at the start of the next round.

B Experiment Configuration

Roles. Every run is driven by an LLM configuration with three roles: the *generator* (candidate and final spec generation), the *clarifier* (question generation), and the *proxy* (the simulated user). The clarifier and generator use the backbone under evaluation; the proxy is GPT-4o-mini at temperature 0 in every experiment, so all methods and backbones face the same answer channel.

Backbones. We evaluate GPT-4o-mini, GPT-4.1-mini, Claude Opus 4.6, Llama-4-Scout-17B-16E-Instruct, and Qwen3-Coder NEXT, each accessed through its provider's standard API. The pipeline's prompts are reproduced in the *Prompts* section at the end of this supplement.

Method hyperparameters. Table 6 lists the settings of our method; all runs use the values shown unless noted otherwise. The method has three design knobs: the pool size N, the diverse-sampling temperature T_{div}, and the entropy floor θ. The main paper's sensitivity ablation sweeps the two knobs, $N \in \{5, 10, 20\}$ and $T_{\text{div}} \in \{0.5, 0.7, 0.9\}$, on a 100-task sample (Claude Opus 4.6, K=5), with θ held fixed.

Baselines. All baselines use the same budget K, the same proxy, and the same output protocol (final spec regenerated from the clarified intent). Best-of-N and Self-Consistency generate 10 candidate questions per round, matching our pool size. ATD is evaluated at $K \in \{5, 10\}$ because of its cost (§D.3).

C Evaluation Metrics

We evaluate a generated specification against the ground-truth reference along the two axes of IaC ambiguity: whether the infrastructure has the right **structure**, and whether its concrete **values** are exactly right. Both metrics operate on specs normalized to a common label namespace (§C.2). **Structure** scores the resource/topology graph via a normalized Graph Edit Distance (§C.3); **Res+Attr F1** is a deliberately strict, exact-match score over concrete resource instances and attribute values (§C.4).

C.1 Preliminaries

Spec representation. Each specification s consists of three fields: **Resources** $\mathcal{R}(s)$, a mapping from a label to a Terraform resource address (e.g., `main` $\mapsto$ `aws_vpc.main`); **Topology** $\mathcal{T}(s)$, a mapping from each label to the labels it depends on; and **Attributes** $\mathcal{A}(s)$, a mapping from each label to a key–value dictionary of configuration parameters (e.g., `cidr_block`, `instance_type`). This is the operational form of the triple $s = (R, T, A)$ of Eq. (1).

Resource-type extraction. Given a Terraform address a, the resource type $\tau(a)$ is its provider-type prefix: `aws_vpc` for `aws_vpc.main`. For data sources the `data.` prefix is part of the type (e.g., `data.aws_iam_policy_document`).

C.2 LABEL NORMALIZATION

LLMs choose arbitrary instance names (`main_vpc`, `vpc1`, `v`) for the same resource type, so two specs describing the same design may differ only in these names. Such syntactic variants should compare as equal; architecturally distinct configurations should not. We therefore apply a deterministic label normalization to both the generated and the reference spec before scoring.

Procedure. Given a spec, we (1) coerce every resource address to a string (some models emit an address as a singleton list, e.g., `["aws_vpc.main"]`); (2) group resource labels by their extracted type $\tau(\cdot)$; (3) sort the labels within each type group, for determinism; and (4) reassign each label a type-indexed name (e.g., `aws_vpc_0`, `aws_subnet_0`, `aws_subnet_1`), rewriting the address accordingly (`aws_vpc.aws_vpc_0`; data sources keep their `data.` prefix). Topology dependencies and attribute maps are re-keyed through the resulting label map. The output is a spec whose label set depends only on the resource types present and their multiplicities.

Effect on each metric. Both metrics compare resource types rather than instance names—GED matches nodes by type, and Res+Attr F1 keys every item by type—so the scores are already invariant to renaming. Normalization canonicalizes malformed addresses, fixes one deterministic label scheme for both specs, and keeps the node alignments reported by GED readable.

C.3 STRUCTURE SCORE (GED)

Graph construction. Each normalized spec is converted to a labeled directed graph $G = (V, E)$: each resource label becomes a node v with $\mathrm{type}(v) = \tau(\mathcal{R}[\ell])$, and each topology dependency $(\ell \to d)$ becomes a directed edge, added only when both endpoints exist as nodes.

Cost model. We compute the Graph Edit Distance between the reference graph G_{ref} and the generated graph G_{gen} under unit costs: node and edge insertions and deletions cost 1; substituting a node by one of the same type is free, by one of a different type costs 1; edge substitutions are free, as edges carry no labels. This penalizes missing or extra resources and incorrect wiring in a single quantity, unlike type-count schemes that score resources and topology independently.

Computation. GED is computed with NetworkX's anytime edit-path search; we keep the best path found within a 30-second per-task timeout. The distance is bounded above by $D = |V_{\text{ref}}| + |E_{\text{ref}}| + |V_{\text{gen}}| + |E_{\text{gen}}|$, the cost of deleting every reference element and inserting every generated one.

Normalized score. The raw distance is normalized by the total graph size D and inverted:

$$S_{\text{struct}} \;=\; \max\!\left(0,\; 1 - \frac{\mathrm{GED}(G_{\text{ref}}, G_{\text{gen}})}{D}\right), \tag{3}$$

so $S_{\text{struct}} \in [0, 1]$, with 1.0 an exact structural match; two empty graphs score 1.0. The score reads as the fraction of the maximal structural distance that the generation closes—a score of 0.55 means just over half the distance to the reference is closed. It measures equivalence to the reference, not deployability; the main paper's deployability ablation measures the latter via `terraform plan` pass rates.

C.4 RES+ATTR F1 (EXACT-MATCH)

Structure captures graph shape but not concrete configuration values. Res+Attr F1 asks the complementary question: how much of the ground-truth spec—its resource instances and its meaningful attribute values—did the generation reproduce exactly? It requires no graph matching; it is computed by occurrence-aware multiset (bag) intersection.

Bag construction. Each normalized spec s is reduced to one occurrence-aware bag $\mathcal{B}(s)$ of items. For every resource label ℓ with address a:

- add one **resource item** $(\mathsf{R},\, \tau(a))$;

- flatten the attribute map $\mathcal{A}(s)[\ell]$ to dotted-key/value leaves (below), and for each retained scalar value under key k, add one **attribute item** $(\mathsf{A},\ \tau(a),\ k,\ \hat{v})$, where $\hat{v}$ is the canonicalized value.

Because the bag is a multiset, multiplicity counts: three `aws_subnet` instances in the reference but two in the generation yield at most two matched resource items and one missing. Attribute items are keyed by resource type, so an attribute value can only match if its resource type was produced—attribute credit is conditional on resource coverage.

Attribute flattening. Attribute values are flattened to `dotted.key` → scalar (or list of scalars) leaves. On the reference side, whose attributes are extracted from the reference program's `terraform plan` output, expression wrappers are unwrapped (`{constant_value: x}` → `x`) and cross-resource pointers (`{references: ...}`) are dropped: references encode topology, which Structure already scores. Nested blocks (lists of dicts) flatten under a shared dotted prefix; a list of scalars is kept as a leaf and later exploded into one item per element; `null` leaves are dropped on both sides.

Value canonicalization. Each scalar is canonicalized before matching: booleans map to `bool:<true|false>`, numbers to `num:<float>` (so `1`, `1.0`, and `"1"` agree), and any other value to its case-folded, whitespace-stripped string. There is no embedding or fuzzy credit: values that are close in form can be incompatible in behavior (`10.0.0.0/16` vs. `10.1.0.0/16` do not match).

Exclusions. Two families of items are dropped from both bags because they carry no cross-spec matching signal: (1) *free-form and name-like keys*, identified by a leaf-name denylist—leaf $\in$ {`name`, `description`, `comment`, `tags`, `arn`, `id`} or a leaf ending in `_name`, `_arn`, or `_id`—since user-chosen names, human descriptions, and opaque IDs legitimately differ between a correct generation and the reference; and (2) generated-side *reference interpolations*, i.e. values containing `${...}`, the generated analogue of the reference's dropped `{references}` pointers.

Score. Let $\mathcal{B}_{\text{ref}}$ and $\mathcal{B}_{\text{gen}}$ be the reference and generated bags. The matched count is the multiset intersection

$$m \;=\; \big|\mathcal{B}_{\text{ref}} \cap \mathcal{B}_{\text{gen}}\big| \;=\; \sum_{i} \min\big(\mathcal{B}_{\text{ref}}(i),\, \mathcal{B}_{\text{gen}}(i)\big), \tag{4}$$

and the score is the exact-match F1 of the combined resource-and-attribute bags:

$$P = \frac{m}{|\mathcal{B}_{\text{gen}}|}, \qquad R = \frac{m}{|\mathcal{B}_{\text{ref}}|}, \qquad S_{\text{R+A F1}} = \frac{2PR}{P+R}. \tag{5}$$

An empty generated bag scores 0. Resources and attributes share one bag, so a single F1 rewards both provisioning the right resource instances and setting the right values. The score is not comparable to soft or embedding-based attribute metrics: values must match exactly after canonicalization.

C.5 Scoring Against a Single Reference

Each task has one verified reference configuration—a valid target, not the unique one. Both metrics give partial credit accordingly: Structure is normalized and rewards overlap, so a valid alternative architecture does not score 0 but scores in proportion to how much of the reference graph it reproduces, and Res+Attr F1 credits every exactly-matching resource instance and attribute value. Absolute scores are still bounded by the single-reference design: many valid programs satisfy the same intent, and Res+Attr F1 is intentionally strict about exact values (a different but workable CIDR scores 0 on that item), which depresses absolute numbers for every method. Since all methods are evaluated against the same reference under the same procedure, the relative comparison carries the signal.

D Computational Cost

We report per-task token usage and wall-clock time for every method. Tokens/task is the mean total number of LLM tokens (prompt and completion) consumed per task; Time/task is the mean per-task runtime. The quality scores of these same runs are in the main paper.

Table 7: Per-task cost across five backbone LLMs at $K=5$ (Ambig-IaC, 300 tasks).

Model	Method	Tokens/task	Time/task
Claude Opus 4.6	Ours	35.8k	75s
	Direct Q Gen.	2.5k	16s
	Best-of-N	10.0k	64s
	Self-Consistency	10.6k	43s
GPT-4.1-mini	Ours	35.0k	58s
	Direct Q Gen.	1.9k	7s
	Best-of-N	8.4k	29s
	Self-Consistency	8.8k	24s
Qwen3-Coder NEXT	Ours	37.8k	73s
	Direct Q Gen.	1.9k	7s
	Best-of-N	9.3k	27s
	Self-Consistency	10.0k	20s
Llama-4-Scout	Ours	33.9k	47s
	Direct Q Gen.	1.8k	4s
	Best-of-N	8.2k	15s
	Self-Consistency	8.4k	12s
GPT-4o-mini	Ours	31.8k	28s
	Direct Q Gen.	1.8k	3s
	Best-of-N	8.2k	14s
	Self-Consistency	8.6k	13s

Table 8: Per-task cost (Tokens/task / Time/task) vs. interaction budget K (GPT-4o-mini, Ambig-IaC). *Direct* is the no-clarification reference.

Method	$K=5$	$K=10$	$K=15$
Ours	31.8k / 28s	61.8k / 50s	93.8k / 69s
Direct Q Gen.	1.8k / 3s	2.0k / 4s	2.2k / 6s
Best-of-N	8.2k / 14s	18.0k / 31s	30.0k / 48s
Self-Consistency	8.6k / 13s	25.9k / 38s	55.4k / 91s
Direct ($K=0$)		0.7k / 1s	

D.1 Cost Across Backbone Models

Table 7 gives the per-task cost of our method and the three lightweight baselines on all five backbones at $K=5$, the budget used in the main paper's cross-model comparison. Our method costs more than the three lightweight baselines—the price of maintaining and regenerating the candidate pool—and the gap is similar across backbones. ATD is compared separately in §D.3.

D.2 Cost vs. Interaction Budget

Table 8 shows how cost scales with the question budget $K \in \{5, 10, 15\}$ on GPT-4o-mini. Our token cost grows roughly linearly in K: each round adds one disagreement-driven question and, when the pool is exhausted, a conditioned regeneration. Direct question generation is nearly flat, since each round adds only one short prompt; Self-Consistency grows fastest among the baselines because each round generates, embeds, and clusters a fresh batch of candidate questions.

D.3 Comparison to ATD

ATD scores freely generated candidate questions over a question $\times$ solution grid, so its per-round cost scales with the product of the two; ours scales with the candidate pool alone, since questions are derived from computed diffs rather than generated and re-scored. Table 9 shows what this means in practice on GPT-4o-mini. At $K=5$, ATD spends $4.1\times$ our tokens and $10.8\times$ our wall-clock per task—the $4\times$/$11\times$ figures quoted in the main paper—while trailing our Structure score. At $K=10$, ATD's recorded token figure is inflated by prompt-cache reads; even excluding them it spends $3\times$ our tokens, and its wall-clock remains an order of magnitude higher.

Table 9: Per-task cost of ATD vs. ours (GPT-4o-mini, Ambig-IaC). †Includes prompt-cache reads (∼1.3M cached tokens per task); 188.0k excluding them.

K	Method	Tokens/task	Time/task
5	Ours	31.8k	28s
	ATD	129.1k	303s
10	Ours	61.8k	50s
	ATD	1451.6k†	512s

Table 10: Output-protocol ablation (GPT-4o-mini, K=15, Ambig-IaC). **Ours-BaselineOut** keeps our question selection but emits the final spec via the baseline clarified-intent regeneration. Parentheses give the relative drop from **Ours**.

Method	Struct. (%)	R+A F1 (%)
Ours	**54.84**	**26.53**
Ours-BaselineOut	49.07 (↓10.5%)	26.01 (↓2.0%)
Direct Q Generation	46.31 (↓15.5%)	18.45 (↓30.5%)
Best-of-N	43.12 (↓21.4%)	19.52 (↓26.4%)
Self-Consistency	38.19 (↓30.4%)	16.12 (↓39.2%)

E Output-Protocol Ablation

Our method differs from the baselines in two places: which questions it asks and how it emits the final specification. The baselines share one output protocol—after K rounds, the accumulated Q&A is folded into a natural-language *clarified intent*, and the spec generator produces the final spec from it. Ours instead returns a surviving pool candidate, falling back to a single conditioned generation only when nothing survives. This ablation separates the two.

Setup. **Ours-BaselineOut** keeps our question selection unchanged and replaces only the output stage: after the interaction, the same generator receives the original task and the accumulated Q&A history, constructs a clarified intent, and generates the final specification from it—exactly the baselines' protocol. The comparison is at K=15 on GPT-4o-mini; the largest budget gives the regeneration protocol the longest history to work from.

Results. Table 10 reports the ablation. Ours-BaselineOut still beats every baseline on both metrics, so disagreement-driven question selection helps on its own, independent of the output stage. The full method adds a further gain, mostly in Structure: surviving candidates have passed the generated answer-based keep-set filters, avoiding potential changes introduced by a separate final-generation step, while a spec regenerated from free text has no such protection. Read as a pair, the two gaps separate the contributions: Ours-BaselineOut vs. the baselines isolates question selection; Ours vs. Ours-BaselineOut isolates the output stage.

F Hyperparameter Sensitivity

We vary the pool size $N \in \{5, 10, 20\}$ and sampling temperature $T \in \{0.5, 0.7, 0.9\}$ (Table 11). The spread across all nine settings is small; neither knob shows a monotonic trend, and every cell outperforms the strongest same-backbone baseline from Table 5. Notably, the paper default (N=10, T=0.7) is not even the strongest cell in the sweep, yet it wins everywhere it is compared: the gains come from the disagreement-driven procedure, not from a fragile hyperparameter choice.

G Cross-Axes Balancing Ablation

Our method builds the shortlist presented to the question generator by interleaving per-axis disagreement rankings in round-robin order (round-robin shortlist construction) rather than by raw entropy alone. In this ablation, the shortlist is instead constructed in raw entropy order. We see a perfor-

Table 11: Hyperparameter sweep over pool size N and temperature T (Claude Opus 4.6, K=5).

N	T	**Struct.**	**R+A F1**
5	0.5	67.27	33.03
5	0.7	63.19	33.11
5	0.9	62.24	32.56
10	0.5	63.62	32.85
10	0.7†	64.69	32.31
10	0.9	66.10	33.31
20	0.5	63.69	32.81
20	0.7	64.27	33.14
20	0.9	65.49	34.66

Table 12: Cross-axes balancing ablation (GPT-4o-mini).

K	**Struct.**	**Δ Struct.**	**R+A F1**	**Δ F1**
5	47.18	−1.77 (↓3.62%)	21.08	−0.96 (↓4.36%)
10	49.74	−1.24 (↓2.43%)	23.25	−0.96 (↓3.97%)
15	51.65	−3.19 (↓5.82%)	24.34	−2.19 (↓8.25%)

mance drop on both metrics at every budget (Table 12), largest at K=15 (−3.19 Structure, −2.19 F1), suggesting that round-robin shortlist construction does help with IaC disambiguation.

H Simulated-User Validation

All results rely on a simulated user: GPT-4o-mini at temperature 0, which answers each yes/no question from the reference configuration (Section 5). We audit how accurate and how stable its answers are.

Sample. We draw 50 tasks uniformly at random (seed 0) and collect every question–answer pair asked on them by our method and the three baselines (GPT-4o-mini, K=10), and by our method with Claude Opus 4.6 (K=5). This gives 250 dialogues and 2,250 pairs. The pairs come from the actual runs, so the audit covers the questions each method really asks.

Correctness. A judge (Claude Opus 5) receives the reference configuration, the request, and the questions, but *not* the simulated user's answers. For each question, it gives the answer a Terraform expert would give from the reference alone. It also states how the reference determines that answer: the asked-about component is declared; a different choice is declared; the component is not declared at all; or the question is too vague, compound, or value-dependent to be determined, which we call *ambiguous*.

Stability. We test stability in three ways. (i) We re-ask 200 recorded questions verbatim, on the recorded conversation prefix. (ii) We re-ask the same questions after a meaning-preserving paraphrase. (iii) The judge flags pairs of answers within a dialogue that cannot both hold. (iv) We replay all 2,250 questions with a stronger model as the simulated user (Claude Opus 5, same prompt and reference) and measure how often the answer changes.

Results. Table 13 summarizes the audit. The simulated user answers 93.2% of reference-determinable questions correctly. On the 12% of questions the judge considers ambiguous, its answer is one the judge accepts in 99% of cases. Answers are also stable: 99.0% are reproduced when re-asked, only 4.0% flip under paraphrase, and 4.4% of dialogues contain a contradictory pair. Accuracy is similar across the methods whose questions are answered (Direct Q Generation 91.6%, Best-of-N 89.2%, Self-Consistency 95.6%, ours 95.4% with GPT-4o-mini and 94.9% with Claude Opus 4.6), so differences between methods are not explained by the answer channel. Most residual errors are answers to components the reference declares under different vocabulary (e.g., "a message queue" for an MSK cluster); such errors lower every method's score rather than inflate it. The

Table 13: Simulated-user validation on 2,250 question–answer pairs from 250 dialogues, with 95% task-level bootstrap CIs. The re-ask and paraphrase checks use 200 pairs; the stronger-proxy check replays all pairs.

Check	Result (%)
Accuracy, reference-determinable questions (88% of pairs)	93.2 [91.9, 94.4]
Answer within the judge-acceptable set, all pairs	95.3 [94.2, 96.4]
Questions judged ambiguous	12.0
Verbatim re-ask reproduces the recorded answer	99.0
Answer flips under paraphrase	4.0
Dialogues with a contradictory answer pair	4.4
Same answer with Claude Opus 5 as the simulated user	90.0 [88.6, 91.5]

answers also depend little on which model plays the user: with Claude Opus 5 as the simulated user, 90.0% of answers are unchanged (Cohen's κ 0.74), and on reference-determinable questions, 85% of the changed answers correct an error of GPT-4o-mini (117 of its 135 errors are fixed, while only 20 of its 1,844 correct answers are overturned).

Scope. This audit shows that the simulated user answers accurately and reproducibly with respect to the target configuration, which a controlled comparison of questioning strategies requires. It does not establish behavioral fidelity to real cloud users, who may be uncertain, misread technical questions, or revise their preferences. A study with cloud practitioners is an important next step.

I Failure Analysis

We analyze the tasks on which our method fails, using the Claude Opus 4.6 run at $K{=}5$ (seed 1 of Table 4, the run behind Table 5). A task *fails* if its Structure score is below 50: 76 of 300 tasks. For every failed task, a judge (Claude Opus 5) reads the complete execution trace (request, reference, every candidate pool, every question with its answer and pruning, and the returned specification) and assigns each resource or dependency edge that differs from the reference one category of Table 14, following the order of the pipeline: did the pool ever contain the right option, was the disagreement asked about, was the question answerable, was the answer honored; and, for the two evaluation categories, was the simulated user wrong or is the difference an equivalent encoding. A task's *primary* category is the one accounting for the largest share of its structural error.

The candidate pool is the bottleneck. Four of five failures (rows 1–3, 80.3%) are decided before the first question: the intended option is in no candidate, so disagreement-driven questioning has nothing to ask. Most often the pool converges (row 1): diversity sampling varies the architecture around a common service but rarely reaches a less common one (ECS instead of Elastic Beanstalk, ElastiCache instead of DAX, Lex v2 instead of v1). Almost as often the pool over-scaffolds (row 2): every candidate carries the same default networking and IAM, so the pool agrees on it and the extras are never put to the user. Both are blind spots of a disagreement-based signal. Retrieval-augmented generation (RAG) is the natural fix: seeding the pool with resource types and co-occurrence patterns retrieved from a corpus of Terraform configurations for the request would surface less common services that free sampling misses, and an explicit "do you need X?" pass over pool-wide extras would expose shared scaffolding.

Questioning, output, and evaluation. Compound questions are the largest remaining cause (row 5, 10.5% primary; present in 38% of failed tasks): one yes/no to "IAM roles and log groups?" drops or swaps a resource. A one-resource-per-question constraint is the direct fix. Richer formats (multiple choice, free-form answers) could resolve such questions in a single turn, but they also make the simulated user's answers harder to ground in the reference and harder to control across methods, so we leave them to future work. Disagreements left unasked (row 4) are rare and mostly question-generation aborts, and confirmed resources are dropped from the output in 2.6% of failures (row 6). Simulated-user errors and valid alternatives (rows 7–8) together account for 1.3% within this audited subset, so neither the simulated user nor single-reference scoring drives the observed failures; the audit covers failed tasks only and does not bound their effect across the full evalua-

Table 14: How our method fails on the 76 of 300 tasks with Structure below 50 (Claude Opus 4.6, K=5, seed run 1). A judge (Claude Opus 5) reads the complete trace of every failed task and assigns each missing or wrong element one category; *Primary* is the share of failed tasks whose structural error comes mostly from that category (sums to 100).

Category	What goes wrong	Example (task ID)	Primary (%)
1. Pool converges on the wrong service	All N candidates pick the same common service; the intended, less common one never appears, so no disagreement and no question can raise it	(7) Elastic Beanstalk blue/green intended; every candidate is an ALB with weighted target groups	52.6
2. Pool over-scaffolds	Every candidate adds the same standard networking or IAM (subnets, NAT, EIP, roles) the user did not ask for; because all agree, the extras are never questioned	(279) a bare VPC with an internet gateway intended; all candidates add NAT, EIP, subnets	21.1
3. Pool omits a supporting resource	A required auxiliary resource (IAM policy document, DNS records) is absent from every candidate	(3) Route 53 zone and weighted records; every candidate is RDS only	6.6
4. Disagreement never asked	Candidates disagreed but the budget went elsewhere; in three of the four tasks (15, 55, 98) question generation aborted after 0–2 questions	(62) EKS node group and public routing in the pool; questions go to IAM roles	5.3
5. Compound or either/or question	A single yes/no cannot be mapped to one resource; the answer drops or swaps an element	(40) "IAM roles and log groups?"; "No" drops the IAM role	10.5
6. Confirmed resource dropped	The user confirmed it, yet the returned specification omits or contradicts it	(138) "No" to DynamoDB, yet a DynamoDB table is returned	2.6
7. Simulated-user error	The answer to a well-posed question disagrees with the reference	(46) "No" to an inline policy; the reference uses one	0.0
8. Valid alternative	Functionally equivalent encoding: inline block vs. separate resource, or provider API version (Lex v1/v2)	(140) replicas as `aws_dynamodb_table_replica` vs inline	1.3

tion. No failed specification contains a non-existent resource type (every type is a valid Terraform provider type), so hallucinated resources are not a failure mode.

PROMPTS

This section reproduces the key prompts used in our pipeline, shown as full-width boxes on this and the following pages; text in angle brackets, `<like this>`, marks values substituted at run time.

SPECIFICATION GENERATION

The system prompt and the accompanying user message are shown in the *Specification generation* boxes; `Feedback / issue` is omitted from the user message before the first answer.

CLARIFICATION-QUESTION GENERATION

VERBALIZE (§A.5) sends the *Question generation* system prompt with `{n}` instantiated to 1; the `keep(...)` labels become the keep sets $\mathcal{P}^{\mathsf{yes}}$ and $\mathcal{P}^{\mathsf{no}}$. The diff summary supplied in the user message renders the shortlist F^* per axis (empty sections and ID-like attribute keys omitted).

SIMULATED USER (PROXY)

The proxy (GPT-4o-mini, temperature 0 for every method and backbone) answers strictly from the ground-truth configuration, appended to the system prompt as `Reference configuration: <ground-truth main.tf>`; each question arrives as its own user message.

INTERACTION-HISTORY CONDITIONING

The accumulated Q&A is folded into a single *clarified intent* string used to condition (re)generation (§A.2).

SPEC-TO-TERRAFORM CONVERSION

The deployability ablation converts each method's final spec into a `main.tf` and runs `init` → `validate` → `plan`. Emission user message: `Clarified intent: <clarified intent>` then `Infrastructure spec: <spec JSON>`; failing tasks get one repair pass whose user message adds the failing HCL and the error tail.

Specification generation (system prompt)

```
You are an infrastructure spec generator for AWS Terraform. You are
generating ONE POSSIBLE interpretation of an ambiguous infrastructure
request. The request can be interpreted in multiple valid ways -- your
job is to pick a SPECIFIC interpretation that may differ from others.

Focus on STRUCTURAL diversity -- vary your choices in:
- **Resources**: Consider different AWS services that could fulfill
the same need (e.g., ALB vs NLB, RDS vs Aurora vs DynamoDB, CloudWatch
vs S3 for logging, Lambda vs ECS vs EC2)
- **Topology**: Consider different dependency structures and how
resources connect (e.g., direct vs through a gateway, single-tier vs
multi-tier, which resources reference which)
- **Resource count**: Consider different numbers of resources (e.g., 2
replicas vs 3, single AZ vs multi-AZ)

Output a single JSON object with exactly these keys:

## 'resources':
Maps short label strings to full Terraform addresses in the form
'aws_resource_type.instance_name'. Use real AWS Terraform provider
resource types.

## 'topology':
Maps src_label to list of dep_labels (dependency edges). Labels must
match keys in 'resources'.

## 'attributes':
Maps each resource label to a dict of key Terraform attributes for
that resource.

# Guidelines:
- Focus on structural choices (which resources exist, how they
connect) rather than specific IDs or values.
- Do NOT invent specific IDs, ARNs, subnet IDs, AMI IDs, zone IDs, or
account numbers.
- For attributes that reference other resources, use Terraform
resource references (e.g., "${aws_db_instance.main.id}",
"${aws_route53_zone.primary.zone_id}") instead of hardcoded IDs.
- For attributes, include structurally meaningful ones (e.g., engine
type, instance_class, record type, source_db_instance_identifier)
using resource references where appropriate.
- Include topology edges whenever resources logically depend on each
other.
- Follow the user request and any clarification Q&A, but fill in
ambiguous structural details with specific choices.

Output ONLY the JSON object, no markdown, no explanations.
```

Specification generation (user message)

```
Task description: <request>

Feedback / issue: <clarified intent>
```

```
This is interpretation <i> of <N>. Make specific, concrete choices
that may differ from other interpretations.
```

Question generation (system prompt)

```
You are an AWS infrastructure expert generating clarification
questions.

You are given:
1. An ambiguous infrastructure request from a user
2. A summary of structural differences found across multiple candidate
interpretations (specs)
3. Each candidate spec is identified by an index (0, 1, 2, ...)

Your job: generate {n} human-readable yes/no clarification questions
that resolve the most important structural ambiguities. Each question
must include filter labels showing which spec indices would be KEPT
for each answer.

Rules:
- Questions must be natural, human-readable language -- no JSON, no
terraform syntax, no resource IDs.
- Focus on STRUCTURAL questions: which services to use, how they
connect, how many of each.
- Do NOT ask about specific attribute values like IDs, ARNs, CIDR
blocks, endpoint URLs, names, or regions.
- Each question should discriminate between candidate specs -- if all
specs agree, don't ask about it.
- IMPORTANT: Each answer option (Yes/No) should keep at least 2 specs
when possible. Avoid questions where one answer keeps only 1 spec --
those are too aggressive and waste the question budget.
- IMPORTANT: Questions must be precise and non-overlapping. Do not ask
two questions about the same topic with slightly different wording.
- Label each question with its dimension: [resource], [topology], or
[attribute].

Output format -- output ONLY the questions, no code fences, no extra
text:

[dimension] Question text here?
Yes -> keep(0,2,4)
No -> keep(1,3)

[dimension] Another question?
Yes -> keep(0,1)
No -> keep(2,3,4)
```

Question generation (user message)

```
Infrastructure request: <request>

<diff summary>

Generate exactly 1 clarification question.
```

Diff-summary format (input to the question LLM)

```
=== <N> Candidate Specs ===
Spec 0:
```

```
  Resources: <comma-separated resource types>
  Connections: <src_type -> dep_type, ...>
Spec 1:
  ...

=== Resource Disagreements (specs disagree on whether these exist) ===
- <resource_type>: YES in specs [<ids>], NO in specs [<ids>]

=== Topology Disagreements (specs disagree on these connections) ===
- <src_type> -> <dep_type>: YES in specs [<ids>], NO in specs [<ids>]

=== Attribute Disagreements (structural, non-ID values only) ===
- <resource_type>.<key>: spec <id>: <value>, spec <id>: <value>, ...
```

Simulated user (system prompt)

```
You are an expert in AWS cloud infrastructure and cloud configuration.
You will be given a cloud configuration and true/false questions about
the configuration.
You will need to answer the questions strictly based on the
configuration.

Guidelines:
- Only answer with yes or no.
- You must strictly follow the reference configuration to answer the
question.

Example 1:
Task: Create an AWS resource to help me store images that I want to
display on my website

Reference configuration:
provider "aws" {
  region = "us-east-1"
}

resource "aws_s3_bucket" "example-bucket" {
  bucket = "test-bucket"
}

Question 1: Do you need audit logging for access to the stored images?
Answer: No
Reason: The reference configuration does not contain any audit logging
for access to the stored images.

Question 2: Do you want an S3 bucket for storing the images?
Answer: Yes
Reason: The reference configuration contains an S3 bucket for storing
the images.

Question 3: Do you require any specific backup or recovery options for
the stored images?
Answer: No
Reason: The reference configuration does not contain any specific
backup or recovery options for the stored images.

End of example.
```

Clarified-intent template

```
Task: <original request>

Q: <question 1>
A: <Yes | No>

Q: <question 2>
A: <Yes | No>
...
```

Spec-to-Terraform emission (system prompt)

```
You are an AWS Terraform expert. Given an infrastructure spec
(resources, topology, attributes) and the user's clarified intent,
produce a single main.tf that `terraform plan` will accept.

Requirements:
- Output ONLY valid Terraform HCL. No markdown fences, no commentary.
- Include a terraform { required_providers { aws = ... } } block and a
provider "aws" block (region = "us-east-1" unless intent says
otherwise).
- Use the spec's labels as resource names where reasonable.
- Fill required attributes with sensible defaults; favor a plan-clean
configuration over completeness.
- Do not reference undeclared resources or variables.
```

Error-guided repair pass (system prompt)

```
You are an AWS Terraform expert. The Terraform HCL below failed
`terraform validate` or `terraform plan`. Fix it so both pass.

Requirements:
- Output ONLY valid Terraform HCL. No markdown fences, no commentary.
- Preserve the user's clarified intent and the resource set from the
infrastructure spec where possible.
- Address the specific terraform error shown. Add missing required
attributes, fix invalid references, and drop unsupported/hallucinated
ones.
- Keep the terraform { required_providers { aws = ... } } block and a
provider "aws" block.
```