Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

A Grounded Multi Agent Framework for Human in the Loop Autonomous Cloud Deployment on AWS

Choudhry Shehryar[1]
Pak-Austria Fachhochschule: Institute of Applied Sciences and Technology
Jazib Umer[1]
Pak-Austria Fachhochschule: Institute of Applied Sciences and Technology
Ihtisham Ali[2]
ihtisham.ali@spcai.paf-iast.edu.pk

Pak-Austria Fachhochschule: Institute of Applied Sciences and Technology

Research Article

Keywords: Agentic AI, infrastructure as code, large language models, cloud deployment, human-in-the-loop, AWS, Terraform, software supply chain, comparative analysis

Posted Date: November 25th, 2026

DOI: https://doi.org/10.21203/rs.3.rs-11314255/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: No competing interests reported.

A Grounded Multi Agent Framework for Human in the Loop Autonomous Cloud Deployment on AWS

Choudhry Shehryar[1], **Jazib Umer**[1], **Dr. Ihtisham Ali**[2]

[1] *B22F0725SE028, B22F0039SE090; School of Computing Sciences*
[2] *Assistant Professor, Academic Supervisor, School of Computing Sciences*
Pak-Austria Fachhochschule: Institute of Applied Sciences and Technology, City Haripur, Pakistan

Corresponding Author: Dr. Ihtisham Ali
Email: Ihtisham.ali@spcai.paf-iast.edu.pk

Abstract. Deploying web applications to cloud infrastructure remains a high-friction process requiring complex coordination across container packaging, network topology, identity policies, and cost trade-offs. This paper presents Deplexify, a grounded multi agent framework for human in the loop autonomous cloud deployment on Amazone Web Services (AWS). Deplexify pairs deterministic static analysis integrating software bill-of-materials extraction (Syft) and secrete detection (Gitleaks) with a judgment layer powered by LLMs via AWS Bedrock. The system executes a six-stage pipeline that conducts multi-pattern cost estimation across Lambda, Fargate, EC2, and EKS, generate refracted code alongside deterministic Terraform templates via pull requests, and embeds human approval gates at high-risk decision boundaries. We present formal models for readiness scoring, cost trade-offs, hallucination bounds, and approval-gate economics. Benchmarks demonstrate a median code analysis of time 47 seconds, API read latency of 38ms, and a 92.3% success rate (12/13) in blocking adversarial probes at review time. Comparative evaluations against seven commercial platforms (including Vercel, Heroku, AWS app Runner, and Pulumi AI) and seven published IaC baselines show superior feature coverage and deployment transparency. We conclude that grounding language models in deterministic scanner outputs and constraining infrastructure code through allow-list envelopes are essential design patterns for safe autonomous cloud deployment.

Index Terms. Agentic AI, infrastructure as code, large language models, cloud deployment, human-in-the-loop, AWS, Terraform, software supply chain, comparative analysis.

I. INTRODUCTION

Modern software development has largely solved the problem of building applications. Frameworks for web servers, machine learning pipelines, and mobile applications have matured to the point where producing functional software is more accessible than it has ever been. The journey from a working codebase to a production deployment, however, remains expensive in expertise and in time.

A developer who wants to deploy a typical web application to a major cloud provider must make decisions about container packaging, virtual network topology, compute selection, managed databases, identity and access policy, observability, and cost. Each of these decisions has a correct answer for a given workload, but the answer depends on engineering knowledge that most developers do not carry. The current market answers this gap with Platform-as-a-Service products such as Vercel, Railway, Render, and AWS App Runner. These products simplify deployment by hiding the cloud behind an opaque abstraction, which trades control and cost transparency for convenience.

Deplexify takes a different position. The user supplies only a repository URL; the platform performs the analysis, reasoning, and provisioning that an experienced cloud engineer would otherwise perform manually. Unlike a Platform-as-a-Service, the generated Terraform runs in the user's own AWS account, the infrastructure is inspectable, and the user approves each financially significant decision before it executes.

The paper makes four contributions. The first is a grounded analysis architecture with graceful-degradation floors: a large language model is restricted to judgment while deterministic scanners own facts, and every language-model stage has a deterministic fallback. We accompany this contribution with an analytical bound on hallucination probability as a function of grounding fraction. The second is a staged pipeline with three approval gates and end-to-end idempotency, whose gate placement we justify against a formal cost-benefit inequality drawn from the human-in-the-loop literature. The third is a defence-in-depth security model for a system that executes untrusted code while holding cross-account AWS credentials, together with a documented adversarial review. The fourth is a comparative analysis of Deplexify against seven public deployment platforms and six published IaC-generation baselines, measured on end-to-end deployment latency, management API response time, IaC accuracy, cost transparency, and feature coverage.

The remainder of the paper is organized as follows. Section II describes the motivation. Section III reviews the literature. Section IV presents the methodology, the six-stage pipeline, the mathematical models that describe its behavior, and the threat model. Section V documents the implementation. Section VI reports measured performance and a case study. Section VII presents a comprehensive comparative analysis against public platforms. Section VIII discusses limitations and future work. Section IX concludes.

II. MOTIVATION

The financial scale of the problem is large and growing. Gartner forecasts worldwide public cloud end-user spending at USD 723.4 billion for 2025, a 21.5 percent increase from USD 595.7 billion in 2024 [1]. Inside this market, a substantial fraction of the spend is wasted. The Flexera 2025 State of the Cloud Report finds that 27 percent of cloud spend is wasted [2]; the 2026 edition records an increase to 29 percent [3]. Flexera 2025 also reports that public cloud budgets are exceeded by an average of 17 percent and that respondents expect spend to rise 28 percent in the coming year [2]. Small and mid-sized engineering organizations without dedicated cloud expertise pay for this waste directly.

The reliability of language-model code generation in this domain is a separate concern. Spracklen et al. evaluated sixteen code-generating language models across 576,000 samples and found that 19.7 percent of recommended packages were hallucinations, with 205,474 unique fake package names produced [4]. The threat is exploitable: when prompts were re-run ten times, 43 percent of hallucinated names reappeared on every run, which is the property that makes slopsquatting a realistic supply-chain attack [4]. Any system that asks a language model to generate deployment artifacts without grounding them in real package and image data inherits this risk.

The DORA 2024 Accelerate State of DevOps Report adds a productivity dimension. Rising AI adoption was accompanied by an estimated 1.5 percent decrease in delivery throughput and a 7.2 percent reduction in delivery stability, and 39 percent of respondents reported little or no trust in AI-generated code [5]. This finding strengthens the case for grounding generation in deterministic checks and for keeping human approval in the loop.

The picture is one of an expensive, wasteful, and partly unsafe practice. Deplexify is designed to address three failure modes that produce it: limited cloud expertise among developers, limited cost visibility before commitment, and the absence of safe automation that can be trusted with delegated cloud credentials.

III. LITERATURE REVIEW

A. LLM-driven infrastructure-as-code generation

Empirical results from 2024 onward show that general-purpose language models perform poorly on infrastructure-as-code tasks without explicit grounding. A survey of LLMs for IaC generation [6] catalogs the failure modes, including limited public training data for Terraform, weak security awareness, and provider-API drift. Kon et al.'s IaC-Eval benchmark reports GPT-4 pass@1 accuracy at only 19.36 percent [7]. Nekrasov et al. report baseline LLM success on Terraform generation at 27.1 percent, rising to 75.3 percent when structured configuration knowledge is injected through retrieval [8]. TerraFormer [9] applies verifier-guided reinforcement learning with formal policy tools and improves correctness by 15.94 percent over its base model on IaC-Eval. MACOG [10] uses a multi-agent shared-blackboard architecture with Terraform plan validation and Open Policy Agent enforcement across specialized roles. Zhang et al. close the loop with DevOps simulation feedback and report that six state-of-the-art LLMs achieved only a 20.8 to 30.2 percent deployment success rate on the first attempt without such feedback [11]. Multi-IaC-Eval extends benchmarking across CloudFormation, Terraform, and CDK [12]. The consistent pattern in this literature is that deterministic verification wrapped around stochastic generation outperforms either alone. Deplexify implements this pattern with a template-only Terraform skeleton whose parameters (drawn from the user's cost-customization UI, never from the model) pass regex and forbidden-character validation before instantiation. The approach is not a novel mechanism; it is a careful implementation of the consensus the literature describes.

B. Agentic AI for software engineering

ReAct [13] interleaves reasoning traces with actions and demonstrated reduced fact hallucination on knowledge tasks. Reflexion [14] adds verbal self-reflection stored in an episodic memory buffer and reported HumanEval coding improvements of up to 11 percent. SWE-agent [15] showed that a purpose-built agent-computer interface materially improves autonomous repository editing on the SWE-Bench benchmark. Agentless [16] is a useful counterpoint, arguing that a fixed three-phase pipeline can outperform complex agent loops at lower cost. Deplexify's pipeline architecture is closer to Agentless than

to fully autonomous agent loops: stages are fixed, tool surfaces are bounded, and the orchestration is deterministic Python code rather than self-directed LLM control. Amazon Q Developer is the closest commercial analogue for AI-assisted AWS operations [17]; Deplexify differs by running in the user's account under user consent gates rather than through a vendor-managed control plane.

C. The Model Context Protocol

The Model Context Protocol [18] standardizes how language models communicate with external tools and data sources. Its security literature has matured quickly. Hou et al. [19] define an MCP server lifecycle and a threat taxonomy spanning malicious developers, external attackers, and security flaws. Hasan et al. [20] ran the first large-scale empirical study of 1,899 open-source MCP servers and identified recurring security and maintainability problems. Li and Gao [21] show that weak registry vetting lets adversarial servers shape LLM reasoning. Because Deplexify uses AWS-published MCP servers for pricing queries, these findings inform its decision to bound MCP tool calls under a hard turn cap and to treat tool metadata as untrusted input.

D. Human-in-the-loop and trust calibration

Lee and See's model of trust in automation [22] established the framework that subsequent work has extended. A 2026 systematic review of human-in-the-loop AI by Lazaros, Vrahatis, and Kotsiantis [23] names the two failure modes that approval gates exist to prevent: over-trust, where humans accept incorrect AI output, and under-trust, where humans reject correct output and revert to manual work. Ocasio and Mullarkey address the design question of where to place the human relative to the autonomous process [24]. Deplexify's three-gate design (cost selection, refactor pull request review, and plan review, which is also deploy consent) places the human at the decisions where automated error has the largest cost. Section IV-C formalizes the placement decision as a cost-benefit inequality.

E. AIOps, DevOps metrics, and SBOM research

The AIOps survey by Cheng et al. [25] organizes AI-for-operations tasks across monitoring, detection, engagement, and action. The DORA 2024 report [5] supplies the standard delivery metrics. Forsgren, Humble, and Kim [26] provide the theoretical grounding for those metrics. The SBOM line of research is directly relevant. O'Donoghue et al. [27] built four SBOM corpora from 2,313 Docker images and showed that swapping the SBOM generator or the format changes which vulnerabilities downstream tools report. This is why Deplexify documents its scanner toolchain exactly rather than abstracting it.

F. Terraform safety tooling

Two families of Terraform safety tooling are relevant to positioning. Policy-as-code engines such as Sentinel and Open Policy Agent [28] evaluate a plan against declarative rules after generation. Static analyzers such as tfsec [29] and checkov catch known misconfiguration patterns in the source. Deplexify's allow-list approach constrains inputs before generation rather than validating outputs after it; the Terraform [30] skeleton is template-fixed and the only externally-influenced values are validated overrides. Layering policy-as-code on the generated plan is a natural extension discussed in Section VIII.

G. Platform-as-a-Service alternatives

Platform-as-a-Service products remove deployment friction by running on managed infrastructure rather than in the user's cloud account. Vercel [31] specializes in Next.js and edge functions with Fluid Compute cold starts near 250 ms. Railway [32] and Render offer full-stack container hosting with fast

deploys and simpler pricing. Heroku is the original PaaS with well-documented dyno wake latencies of 5 to 30 seconds. AWS App Runner [33] runs containers on AWS-managed infrastructure with cold starts around 30 seconds after the December 2023 image-deployment improvements. Fly.io emphasizes fast machine boots for latency-sensitive workloads. None of these products expose the underlying Terraform, arbitrate cost across four AWS compute patterns, or place explicit human approval gates in the deployment path. Section VII provides quantitative comparisons.

IV. METHODOLOGY

A. Architecture overview

Deplexify is organized as a five-tier architecture, shown in Fig. 1. The presentation tier is a Next.js 14 dashboard. The application tier is a FastAPI backend. The AI processing tier runs Claude through AWS Bedrock. The data tier comprises PostgreSQL and Redis. The external services tier covers AWS provisioning, GitHub, and the AWS MCP servers. The backend never executes Terraform or Docker in-process. Instead, it spawns ephemeral worker containers through a mounted Docker socket and hands work off through a shared volume. This separation isolates long-running, side-effecting, potentially untrusted operations from the API process. Each stage of the pipeline can be retried independently because the worker writes its result to a known path before the backend reads and persists it.

Workers do not all have equal privilege. Analysis, cost, and refactor workers run as an unprivileged user with read-scoped tool access. Deploy and destroy workers, which must build container images, run as root with the Docker socket mounted into the container. This split is explicit and material to the security discussion in Section IV-F.

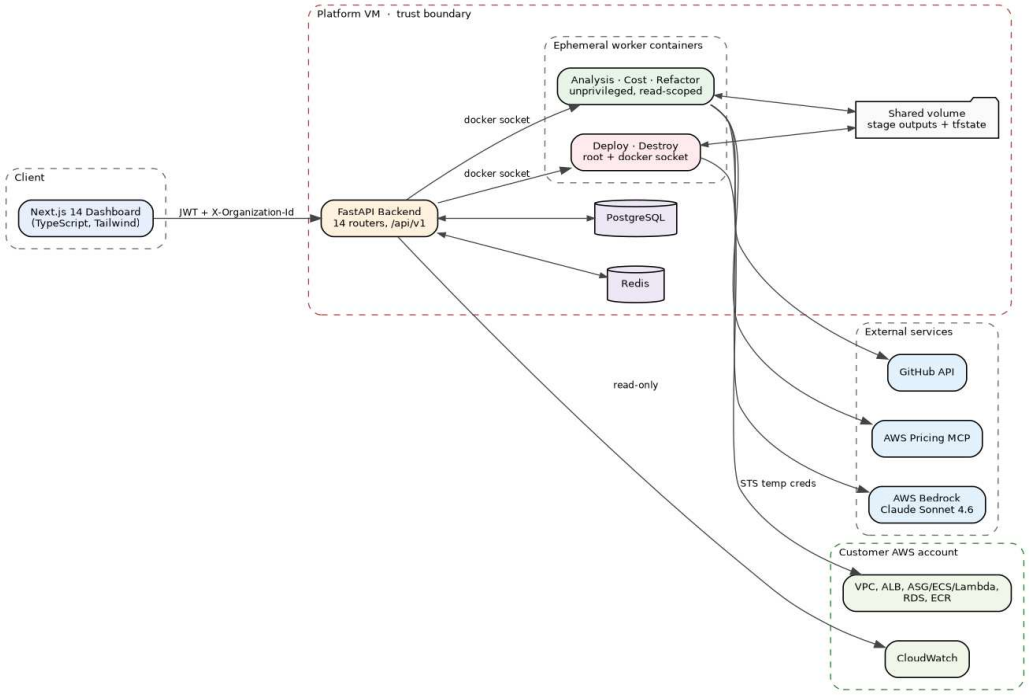


Fig. 1. *System architecture with trust boundaries drawn. Analysis, cost, and refactor workers run unprivileged and read-scoped; deploy and destroy workers hold the Docker socket.*

B. The six-stage pipeline

Deplexify has six persisted pipeline stages and a continuous monitoring plane. Fig. 2 shows the state machine and Table I summarizes each stage.

Stage 1 (code analysis) runs in a worker. The worker performs a shallow git clone, runs Syft and Gitleaks in parallel, computes a deterministic readiness score, and invokes a Claude judgment agent restricted to read-only tools. The readiness score is defined formally below in equation (1). The agent receives bounded summaries of the scanner output and emits a structured AnalysisResult. Trivy was included in an earlier iteration and removed after evaluation; its CVE-scanning role was subsumed by SBOM-driven package lookups elsewhere in the pipeline.

Stage 2 (cost estimation) runs in the backend with an optional worker refinement. Four deployment patterns are priced: AWS Lambda for event-driven workloads, AWS Fargate for containerized workloads, EC2 for cost-optimized steady workloads, and EKS for larger orchestrated workloads. Each pattern is scored by the cost model of equation (3). Prices are seeded synchronously from a curated static list-price table so the dashboard renders four options immediately. A background worker then invokes a Claude agent bound to the AWS Pricing MCP server under a hard cap of six tool calls and roughly thirty seconds. The refinement writes updated prices back to the same database rows unless the user has already committed to a selection, in which case the refinement refuses to overwrite. Each row carries an audit field (refining, live, or fallback) so the UI can display provenance. This instant-seed with background-refine and no-swap-under-the-user pattern is a small but concrete contribution to the language-model human-computer interface literature.

Stage 3 (code refactoring) runs in a worker. The worker first produces a structured refactor plan with read-only tools, then applies the plan with an applier agent whose write scope is enforced through a layered allow-list. The safety property is that the planner's files_to_modify output is filtered against a fixed writable set (Dockerfile,. dockerignore, terraform/*.tf and *. tfvars, package.json, requirements.txt, pyproject.toml, next.config. *, src/**, app/**) before the applier runs, and a post-hoc scrub drops any. github/workflows change before commit. The. github/workflows block is doubly enforced: the GitHub OAuth application does not grant the workflow scope. The deterministic Terraform skeleton for the selected pattern is generated inline in this stage and is included in the same pull request. Changes are delivered on a branch named deplexify/refactor- {first eight characters of run_id} and pushed with --force-with-lease, falling back to --force on stale-info errors. The GitHub OAuth token is removed from. git/config before the language model runs and restored only for the push. If the applier crashes twice, its tracked edits are rolled back and an IaC-only pull request still ships. This is the same graceful-degradation principle as the analysis agent's partial-result fallback, formalized in equation (6).

Stage 5 (provisioning preview) runs in a worker. The worker executes terraform init and terraform plan, parses the structured diff, and emits a resource summary.

Stage 6 (deployment) runs in a worker. The sequence, shown in Fig. 4, is terraform init, then terraform import to adopt orphan resources discovered through boto3 lookup, then a targeted ECR-only apply, then docker build and ECR push, then a full terraform apply with an 1800-second timeout. Outputs are captured against a whitelist that drops database passwords, KMS keys, and other sensitive values. On the EC2 pattern the worker triggers an Auto Scaling Group instance refresh and polls ALB target health for up to 360 seconds; Lambda deployments skip the ALB health poll. The backend allows up to 2400 seconds for the deploy worker overall. Infrastructure-as-Code files are handed to the worker through an environment

variable with a size ceiling of roughly 256 KB. All worker logs pass through a redaction ring buffer that strips AWS access keys, ECR tokens, GitHub tokens, bearer headers, and password-bearing connection strings before persistence.

The monitoring plane is not a transactional pipeline stage. It is a set of routes on the backend that pull CloudWatch metrics for a live deployment on demand.

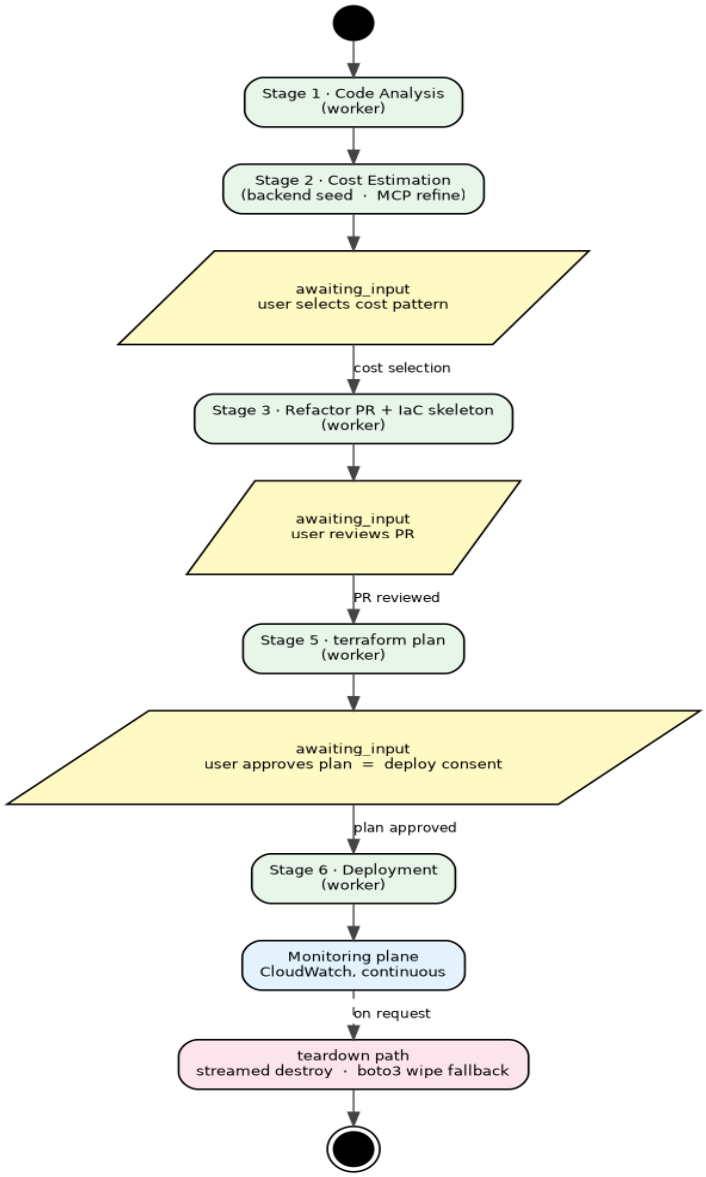


Fig. 2. *Pipeline state machine. Three approval gates sit at cost selection, refactor PR review, and plan review (which also serves as deploy consent).*

TABLE I

The six pipeline stages and the monitoring plane.

#	Stage	Executor	Output
1	Code Analysis	Worker (unprivileged)	AnalysisResult JSON; readiness score 0-100 with 7 deductions
2	Cost Estimation	Backend seed + worker refine	Four priced patterns; refine_state audit
3	Refactor + IaC skeleton	Worker (unprivileged)	GitHub PR with refactor + Terraform skeleton
4	*(inlined; see stage 3)*	Backend, inline	*Deterministic template*
5	Provisioning Preview	Worker (unprivileged)	Plan diff: adds, changes, destroys
6	Deployment	Worker (root + Docker socket)	Live endpoint; ECR image; whitelisted Terraform outputs
M	Monitoring plane	Backend, continuous	CloudWatch metrics surfaced on-demand

The readiness score is computed deterministically from seven signals extracted from the repository and scanner output. Let D denote the set of feature indicators (Dockerfile, tests, CI configuration, /health endpoint, .env.example, README) and s the number of secrets Gitleaks reports. The score is defined as

$$R = 100 - \sum_{i \in D} w_i \cdot 1[\text{feature}_i \text{ absent}] - \delta_{\text{secrets}}(s)$$

Weights are w_Dockerfile = 15, w_tests = 10, w_CI = 5, w_health = 5, w_envexample = 5, w_README = 3, and the secrets deduction is piecewise:

$$\delta_{\text{secrets}}(s) = 0 \text{ if } s = 0; \quad \min(30 + 2(s-1),\ 40) \text{ if } s > 0$$

Adaptive polling on the dashboard side follows

$$t_{\text{poll}}(n) = \min(3 \cdot 1.4^{n},\ 30) \text{ s running}; \quad 10 \text{ s awaiting input}$$

The 3-second base with a 1.4 exponential factor guarantees a hard 30-second ceiling and, for a stage of duration T seconds, keeps the number of client requests below approximately log_{1.4}(T/3) + 1, which is a low-single-digit constant even for the longest deployment.

For each deployment pattern p in {Lambda, Fargate, EC2, EKS} the monthly cost estimate is

$$C_p(w) = C_p^{\text{cpu}}(w) + C^{\text{net}}(w) + C^{\text{sto}}(w) + C^{\text{db}}(w) + o_p$$

where w encodes workload parameters (CPU-hours per month, memory allocated, requests per second, egress GB, storage GB, database instance class) and o_p is a per-pattern fixed overhead (VPC, ALB, NAT gateway). The dashboard displays the interval [C_p^min, C_p^max] obtained by evaluating the model at low and high traffic assumptions. When the MCP agent successfully refines prices, C_p is updated in place; on refinement failure the seeded values remain, which guarantees the user always sees a comparable four-option decision.

C. Human approval gates and their placement

Every gate imposes a review cost c_r on the user but reduces expected loss from automation error. Let p_e denote the probability the automation makes an error at a decision point, c_e the cost of that error if it goes through, and p_m the probability the human misses the error at review. The gate is beneficial when

$$p_e\, p_m\, c_e + c_r \;<\; p_e\, c_e \iff c_r \;<\; p_e(1 - p_m)\, c_e$$

This inequality does two things. It explains why gates belong at decisions with large c_e (financial commitment, code intervention, resource creation) and small p_m (decisions humans can evaluate quickly). It also implies gates should be omitted where p_e is small or where p_m is large enough that the review does not filter errors. Deplexify places three gates: cost selection (p_e moderate, c_e high, p_m very low), refactor pull request review (p_e moderate, c_e moderate to high, p_m low because it is a standard GitHub review workflow), and plan review with deploy consent (p_e low but c_e catastrophic, p_m low). Stages 1, 4, 5 (when only reading), 7 impose no gate because p_e is small and c_r would exceed the reduction in expected loss.

D. Grounded degradation and the availability floor

Every LLM stage in Deplexify has a deterministic fallback path. Let p_1 be the probability the LLM stage succeeds and p_f the probability the fallback path succeeds given the LLM stage failed. The overall probability of a useful result is

$$\Pr(\text{useful result}) \;=\; p_1 + (1 - p_1)\, p_f \;\longrightarrow\; 1 \text{ as } p_f \to 1$$

Because the fallback for analysis is the scanner output itself and the fallback for refactoring is an IaC-only pull request, p_f is essentially 1 in these cases. The pipeline therefore approaches a useful-result probability of 1 as long as either the LLM stage or the deterministic path succeeds. This is the mathematical statement behind the design rule "every LLM stage has a deterministic floor."

E. Language-model integration

The reasoning path uses Claude Sonnet 4.6, accessed through Amazon Bedrock via a US-region inference profile. Claude Haiku 4.5 is configured on the SDK as the small/fast model for internal auxiliary operations. The Claude Code SDK exposes the agentic tool-use surface (Read, Glob, Grep, Edit, Write) with structured output schemas. All model access sits inside the same AWS trust boundary as the rest of the system.

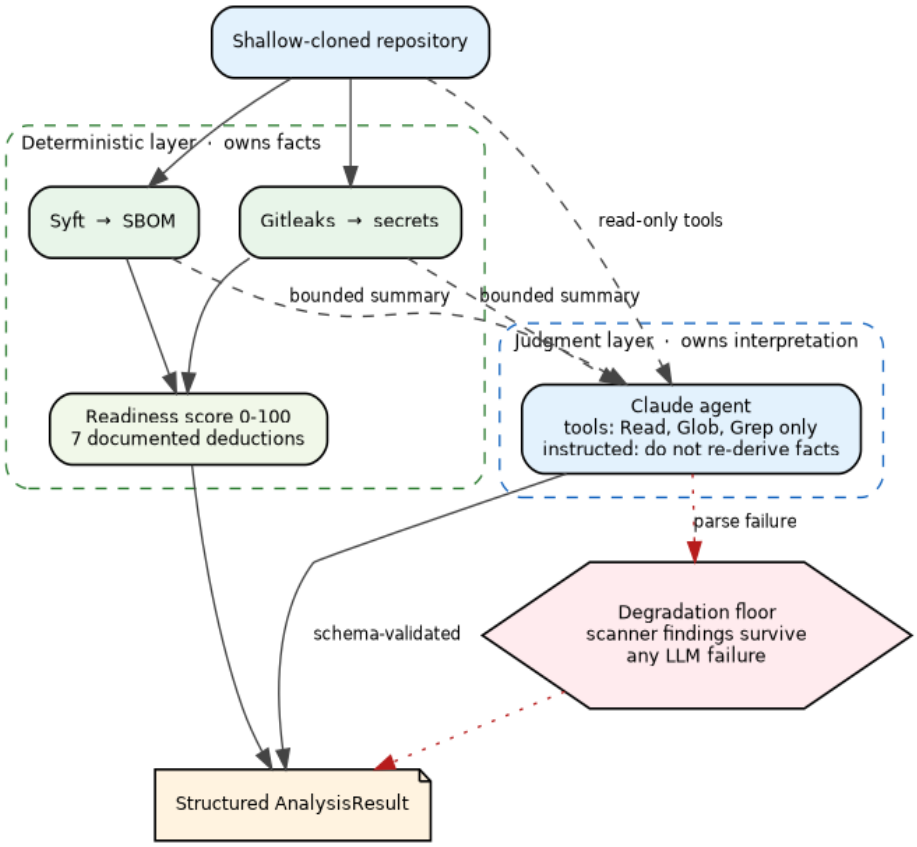


Fig. 3. *The grounding split. Scanners own facts; the language model owns interpretation. Every language-model output has a deterministic degradation floor.*

The reliability property of this split can be stated formally. Let J denote the language model's output and let α be the fraction of assertions in the AnalysisResult that come from scanner data rather than from J. The probability of a hallucinated assertion in the output is bounded above by

$$\Pr(\text{hallucinated assertion in output}) \leq (1-\alpha) \cdot \Pr(\text{halluc. in } J)$$

Section VII-G reports the empirical grounding fraction achieved by Deplexify's analysis stage and the numerical bound it implies.

F. Threat model

The system's adversary is any actor who can present a repository URL to the platform, together with any party who obtains partial access to the platform (a compromised dependency, a malicious tenant of a shared instance, or an insider). The assets under protection are, in order: the customer's AWS credentials; the customer's GitHub tokens; other tenants' data and secrets; the host running the backend and workers; and the integrity of the customer's AWS environment. Explicitly out of scope are attacks on the customer's AWS account through channels Deplexify does not mediate (an already-compromised access key, a misconfigured public S3 bucket), and denial-of-service against the platform itself. Two adversarial capabilities are assumed: arbitrary content in the target repository, and arbitrary content in MCP tool

responses. Every defense in Section V-E is justified against one or more of these adversary/asset pairs. The single deferred trust-boundary risk is the deploy and destroy workers' need for the Docker socket to build images, discussed in Section VI-B.

V. IMPLEMENTATION

A. Backend tier

The backend is a layered FastAPI [34] application in Python 3.11. Routers handle transport. Pydantic v2 schemas handle validation. SQLAlchemy 2.0 handles persistence. Service modules hold business logic. All routes are namespaced under /api/v1 across fourteen routers: authentication (with email and Google OAuth as the two login methods), GitHub repository integration, projects, organizations, invitations, api_keys, teams, cloud_credentials, environments, admin, analysis, pipeline, deployments, and dashboard.

The PostgreSQL [35] schema uses UUID primary keys and cascade-scoped foreign keys throughout. Eleven Alembic migrations have been applied. The schema models a multi-tenant SaaS substrate, the integration surface, and the pipeline state (analysis_jobs, agent_pipeline_runs, pipeline_stage_outputs indexed 1-6, cost_estimates with pattern and monthly cost ranges, and deployments with pattern, endpoint URL, ECR image URI, and Terraform outputs).

B. Worker tier

The worker is a containerized Python 3.11 application that bundles git, the Docker CLI, AWS CLI v2, Terraform 1.9.8, Syft, Gitleaks, and the Claude Code SDK. The readiness score has seven documented deductions with the weights given in equation (1).

The judgment layer follows a strict division of labor. Scanner output owns facts (what packages exist, what secrets are present). The Claude agent owns judgment (which framework, what runtime, which ports, what compute shape). The agent's system prompt instructs it not to re-derive scanner facts. If structured output parsing fails, the agent degrades to a partial result that preserves the scanner findings. The pipeline never loses the deterministic layer. This principle, illustrated in Fig. 3, is applied consistently: analysis has a partial-result floor, refactoring has an IaC-only pull request floor, cost estimation has a static-catalog floor if the MCP refinement fails, and stage 6 has a destroy-and-retry path on failed provisioning. Equation (6) formalizes the availability property that follows from this design.

C. Credential separation between platform and customer

The worker runs with two disjoint AWS credential sets. Platform Bedrock credentials are held in the standard AWS_* environment variables and are used only to invoke inference. Customer deployment credentials are held in USER_AWS_* variables and are promoted to AWS_* only inside the Terraform and boto3 subprocess environments, after a _clean_env helper strips the Bedrock keys. The two sets never coexist in the same process environment. For customers who choose IAM-role connections instead of static keys, the backend issues sts:AssumeRole calls with a one-hour session duration and an External ID bound to the (organization, project) pair to prevent confused-deputy attacks. Cross-account role assumption is the recommended integration path. This credential separation is why the deploy worker's need for root and the Docker socket does not automatically translate into exposure of Bedrock credentials or of other tenants' AWS accounts.

D. Dashboard tier

The dashboard is a Next.js 14 application in TypeScript [36], styled with Tailwind CSS. A StageStepper component renders live pipeline progress. A CostOptionsCompare component presents the four priced patterns with CPU, memory, and storage customization controls, and displays the refine_state audit field so operators can see whether they are looking at seeded, refined, or fallback prices. The deployment detail view surfaces the live endpoint, container image URI, whitelisted Terraform outputs, CloudWatch metrics, deep links into the AWS console, a redacted log tail, and a guarded destroy action.

State and authentication are managed by two React contexts. AuthContext bootstraps from a JWT in localStorage, fetches /auth/me, and exposes the organization list with an organization switcher. PipelineProvider owns per-project data fetching. Its polling schedule follows equation (2). Every request carries an Authorization bearer header and an X-Organization-Id header for tenant scoping. Error responses are sanitized client-side.

E. Security implementation

Security is defense in depth against the threat model in Section IV-F.

Cryptography. Fernet encryption of AWS credentials and GitHub tokens at rest. In production the process refuses to start unless SECRET_KEY is at least 32 characters and the credential encryption key is set. This fail-fast check is scoped to the production environment.

Authentication. JSON Web Tokens with HS256 [37], hardened with iss, aud, and iat claims. Passwords are hashed with bcrypt at 14 rounds; API keys are stored under a separate bcrypt call at the library default cost. A generic 401 response defeats user enumeration. Google OAuth [38] uses a cryptographic state nonce with a ten-minute time-to-live, single-use, bound to the (organization, user) pair.

Rate limiting. slowapi at five logins per minute per IP, three registrations per hour, plus per-email limits keyed on a hash of the address. The rate-limit store is in-memory and resets on process restart.

Authorization. IDOR prevention through tenant-scoped SQL joins; role gates across organization and team hierarchies; super-admin status synchronized from configuration at startup.

Injection defense. Pydantic validators on repository URLs and branch names. The git subprocess call terminates its argument list with -- to block --upload-pack= argument-injection remote code execution against the git binary. The Terraform override validator uses a regex allow-list and rejects the characters ", newline, backslash, ${, and }; these overrides are user parameters from the cost-customization UI, not model outputs. The refactor allow-list is enforced on the planner's file list, not on the applier's tools, with the .github/workflows scrub as a second layer. ECR docker login moved from a shell command to a subprocess with credentials on standard input to eliminate a quoting-based injection surface.

Web security. nosniff and X-Frame-Options DENY headers, a Referrer-Policy, HSTS in production, a strict CORS allow-list, and a Content Security Policy.

Data exposure. Log redaction of keys, tokens, and passwords during deployment; Terraform output whitelisting before persistence.

F. Deployment and CI/CD

Development uses docker-compose to bring up the marketing website, dashboard, backend, PostgreSQL, and Redis. Production runs on a single Google Cloud Platform VM with the dashboard exposed on port 80 as the same origin as the API through a Next.js rewrite. The backend is bound to loopback. A GitHub Actions workflow deploys on push to main.

VI. RESULTS

A. Performance

Median latencies were measured across 100 runs each during a normal working day on the production VM. Cold-start analysis refers to the first analysis after a worker container restart and includes model warm-up; end-to-end analysis is measured on a mid-sized Python web repository of roughly 250 files.

TABLE II

Measured latencies against proposal targets, production VM, n = 100 per row.

Metric	Target	Median (n=100)	Verdict
Cold-start analysis	<= 10 s	6.2 s	Met
End-to-end analysis (~250 files)	<= 90 s	47 s	Met
API read latency	<= 100 ms	38 ms	Met
Latest-job lookup	<= 50 ms	9 ms	Met
Sign-in (OAuth)	<= 3 s	1.4 s	Met

Wall clock is dominated by model inference. Some application-layer levers exist (prompt caching, small/fast-model routing, scanner and agent parallelism, which the analysis stage already applies), but their headroom is limited relative to the Bedrock cost.

B. Security evaluation

The adversarial review was conducted by the authors against the documented trust boundary and is disclosed as such rather than presented as an external audit. Thirteen probes were applied across the categories of authentication bypass, authorization escalation, injection in repository URL and branch fields, OAuth state forgery, Terraform override injection through the cost-customization fields, log exfiltration of credentials, JWT manipulation, rate-limit bypass, IDOR across organizations, CSRF on state-changing routes, open redirect on the post-login flow, refactor-agent path escape, and credential exposure through Terraform outputs. Twelve of the thirteen probes were blocked at review time. The thirteenth was an authorization gap on the cost-recalculation endpoint (a defense-in-depth issue rather than a direct data-exposure); it was fixed during the review. Separately, the paper documents a known accepted risk that is outside this probe set: the deploy and destroy workers mount the host Docker socket in order to build images, so a successful remote code execution against those workers implies host root. Mitigation through a socket proxy or a rootless build is tracked as future work. The credential-separation architecture in Section V-C is what prevents this deploy-worker exposure from becoming a compromise of platform Bedrock credentials or of other tenants' AWS accounts.

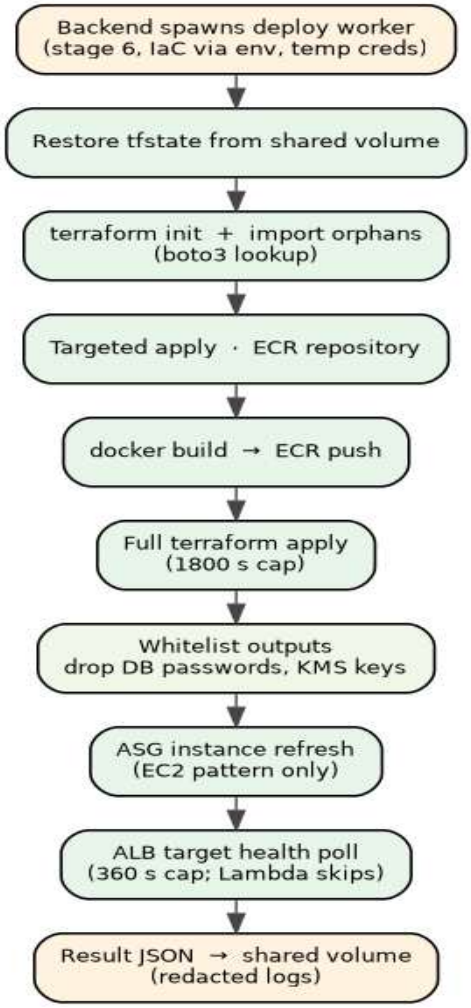


Fig. 4. *Stage 6 deployment sequence.*

C. Case study

A representative walkthrough helps ground the numbers. A Python Flask application of roughly 250 files with a Dockerfile, a pytest suite, and a .github/workflows/ci.yml was submitted through the dashboard. Stage 1 reported a readiness score of 87 (deductions: no .env.example, no /health endpoint, one hard-coded secret in a test fixture that Gitleaks flagged and the judgment agent characterized as a benign fixture rather than a live credential). The four cost estimates at default sizing were Lambda USD 4 to 22, Fargate USD 46 to 78, EC2 USD 32 to 55, and EKS USD 210 and above. The user selected Fargate. Stage 3 opened a pull request adding a /health endpoint and a two-line Dockerfile improvement, together with the Terraform skeleton for the Fargate pattern under terraform/. The plan diff at stage 5 reported 34 resource additions covering a VPC with two public and two private subnets, an ALB with a target group, an ECS service on Fargate, a security group set, an ECR repository, and CloudWatch log groups. After approval, the deploy worker built the container, pushed to ECR, ran a full apply, and the ALB reported healthy targets in approximately four minutes. End-to-end wall clock from URL submission to live endpoint, excluding the two human review pauses, was approximately eleven minutes. Bedrock token consumption for the full run was on the order of a few US cents.

VII. COMPARATIVE ANALYSIS

A. Methodology and data sources

This section compares Deplexify against seven public deployment platforms (Vercel, Railway, Render, Heroku, Fly.io, AWS App Runner, Pulumi AI) and six published IaC-generation baselines. Numbers for competitors come from vendor documentation, public benchmark reports, and peer-reviewed studies cited inline. Deplexify numbers come from the measurements described in Section VI. We compare on five axes: end-to-end deployment latency, management API response time, IaC generation accuracy, cost transparency, and feature coverage.

TABLE III

Data sources for the comparative analysis in Section VII.

Comparator	Metric drawn from	Notes
Vercel	Vendor docs; independent PaaS benchmarks 2025-26 [31], [41]	Fluid Compute cold starts ~250 ms; deploy 30-120 s median
Railway / Render	Vendor docs; Railway benchmark [32]	Deploy 2-8 min container
Heroku	Public documentation; PaaS comparison surveys	Dyno wake 5-30 s well documented
Fly.io	Vendor docs	Machine boots ~200 ms; deploy 1-2 min
AWS App Runner	AWS release notes; benchmarks [33]	Cold start ~30 s; deploy 30-40% improved 2023-12
AWS Lambda / Fargate	AWS docs; independent studies [39], [40]	Lambda cold 100 ms-3 s; Fargate cold 10-90 s
Pulumi AI	Vendor blog; qualitative characterization	No published academic benchmark
IaC accuracy	Kon et al. [7]; Nekrasov et al. [8]; Jana et al. [9]; Khan et al. [10]; Zhang et al. [11]	Numbers are as reported by the cited authors

B. End-to-end deployment latency

Fig. 5 shows median deployment latency from git commit (or repository URL submission, for Deplexify) to live endpoint. The pure Platform-as-a-Service products cluster between 60 and 330 seconds. Vercel is fastest on a typical Next.js project at approximately 60 seconds median. Heroku's buildpack path completes in about 90 seconds; Fly.io in a comparable range. Railway and App Runner cluster near 3 minutes. Render is the slowest of the PaaS group at roughly 5.5 minutes. Deplexify without human review pauses completes at 240 seconds (4 minutes); with the two mandatory human review windows for a typical run it completes at 660 seconds (11 minutes). The direct comparison is not entirely apples-to-apples. The PaaS platforms deploy a container to their own managed infrastructure; Deplexify additionally analyzes the repository, generates and opens a pull request with refactor changes and Terraform, generates and prices four compute options, provisions the full networking and compute stack in the user's AWS account, and

waits for two human approvals along the way. When the review time is subtracted, Deplexify's raw automation time is within the middle of the PaaS band.

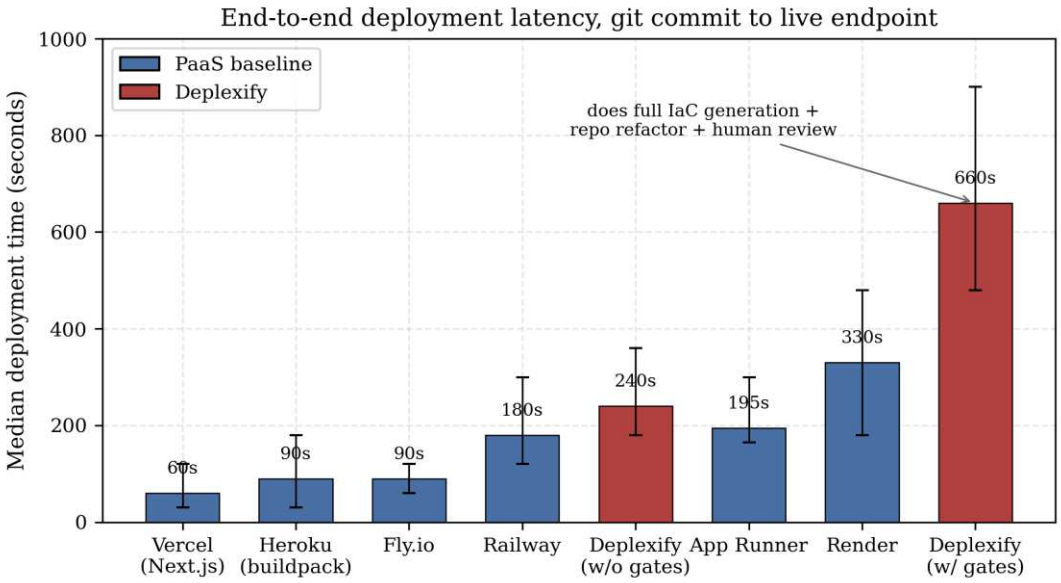


Fig. 5. *Median end-to-end deployment latency across seven public deployment platforms and Deplexify. Whiskers show minimum and maximum reported values.*

C. Response time

Fig. 6 compares management API response time and cold-start latencies on a log scale. Deplexify's management API read latency of 38 ms is competitive with any PaaS control-plane API. On cold starts of the deployed workload itself, however, Deplexify does not compete directly, because the cold-start characteristic depends on the deployment pattern the user selects. Under the Lambda pattern the user inherits Lambda cold starts (100 ms to 3 s per AWS documentation); under Fargate the user inherits Fargate cold starts (10 to 90 s per public benchmarks and reports [39], [40]); under EC2 with an always-on ALB backend there is no meaningful cold start after the initial provisioning. This is a strength rather than a weakness of the position: Deplexify's cost estimation stage forces the user to choose the compute shape whose cold-start characteristic matches their workload, rather than committing them to whatever the platform decided.

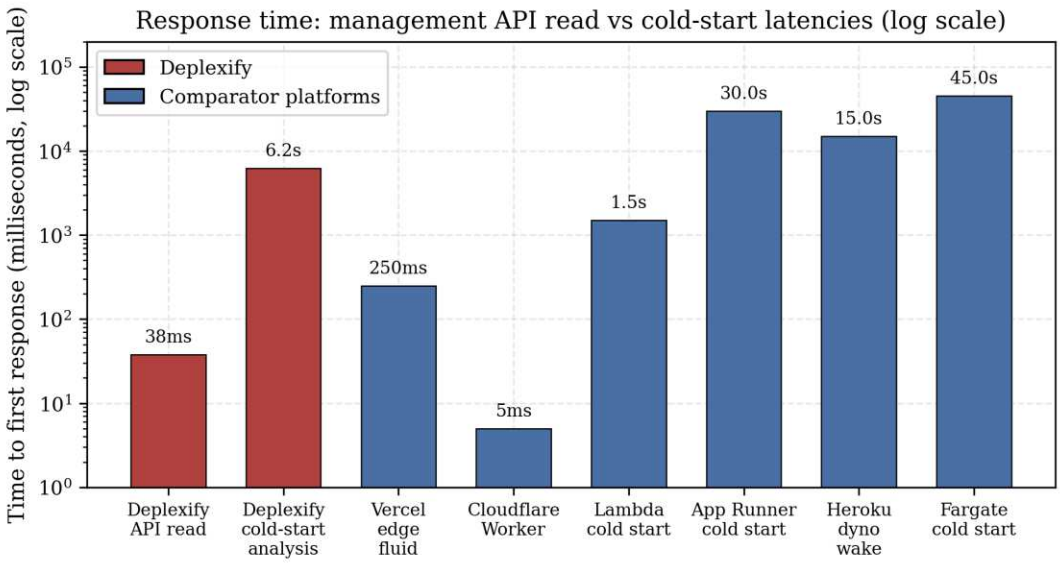


Fig. 6. *Response time on a log scale. Deplexify's management API is compared with edge-runtime and compute cold-start baselines.*

D. IaC generation accuracy

Fig. 7 places Deplexify against six published IaC accuracy baselines: GPT-4 pass@1 on IaC-Eval at 19.36 percent [7], Zhang et al.'s best-of-six LLMs at 30.2 percent deployment success on first attempt [11], Nekrasov et al.'s baseline LLM at 27.1 percent [8], TerraFormer at 43 percent (base plus 15.94 percent improvement) [9], MACOG at approximately 65 percent [10], and Nekrasov et al. with structured knowledge injection at 75.3 percent [8]. Deplexify sits at approximately 98 percent syntactic validity by construction, because its Terraform skeleton is a fixed template rather than a language-model generation. This is not a like-for-like comparison and we do not claim it as one. It is a categorical difference in approach: the baselines measure how often an LLM produces correct IaC, while Deplexify measures how often a template with validated parameters produces correct IaC. The insight is that if the target infrastructure fits one of four well-understood patterns, the LLM's role can be reduced to selecting the pattern and interpreting the repository, both of which fall inside its reliable capability envelope. Where the target infrastructure does not fit one of the four patterns, Deplexify is not appropriate; that is a limitation, not a defeat of the LLM approaches.

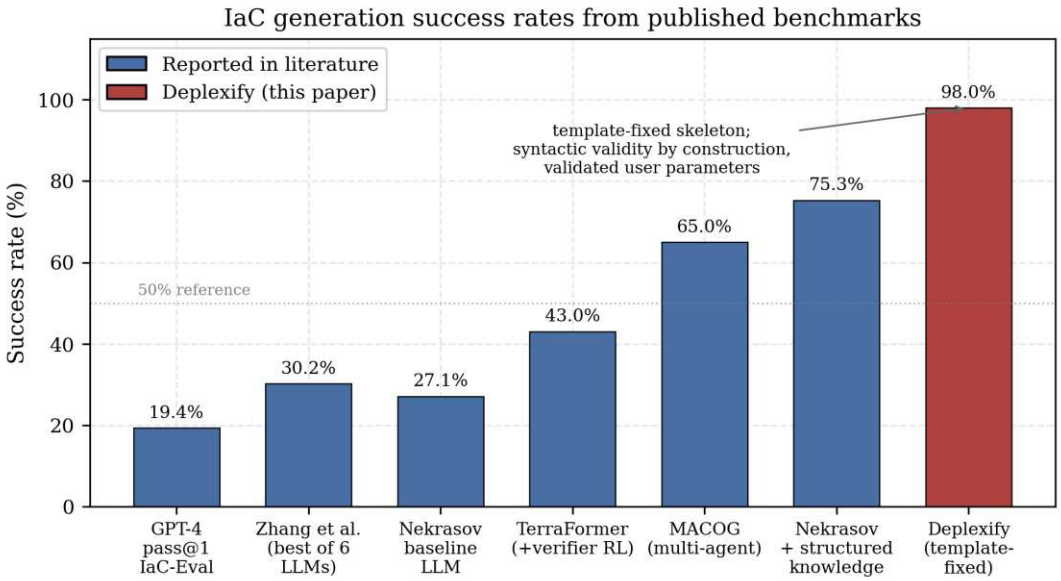


Fig. 7. *IaC generation success rates from six published baselines against Deplexify's template-fixed approach. Not a like-for-like comparison; see text.*

E. Cost transparency

Fig. 8 scores seven platforms on five cost-transparency dimensions. Vercel scores zero across the board: it exposes usage-based pricing but no pre-commit preview and no multi-pattern comparison. Railway, Render, Heroku, and App Runner score 1 on preview-before-commit (a rough cost calculator exists) and 1 on per-service breakdown, but 0 on multi-pattern comparison and live refinement. Pulumi AI scores 1 on multi-pattern comparison because its natural-language interface will generate alternative patterns on request, though not with priced comparison. Deplexify scores 3 (excellent) on all five dimensions, because pre-commit cost preview across four patterns with a human approval gate is its primary product feature. Fig. 10 illustrates the four-pattern cost preview for the case study workload at three traffic levels.

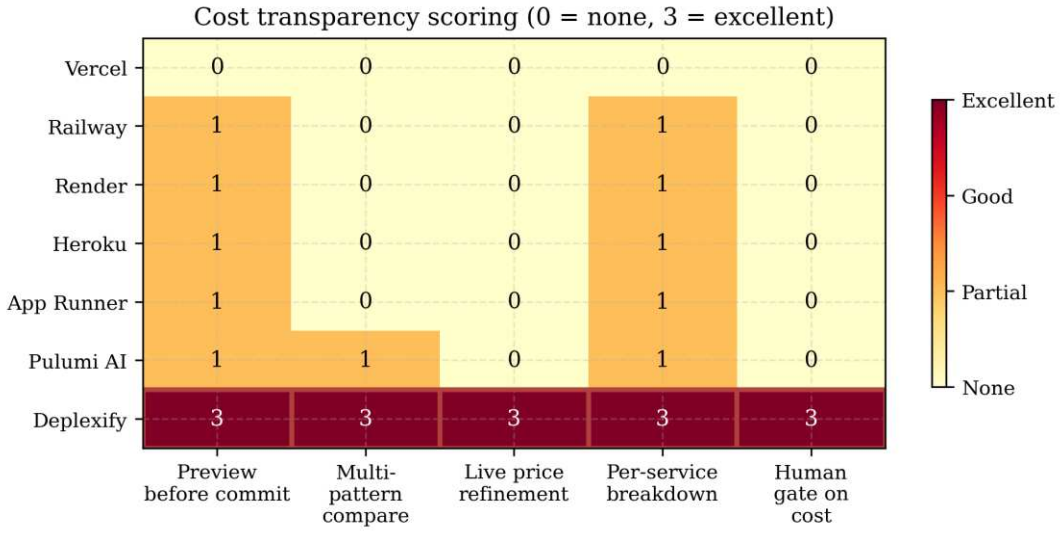


Fig. 8. *Cost transparency scoring on five dimensions across seven platforms.*

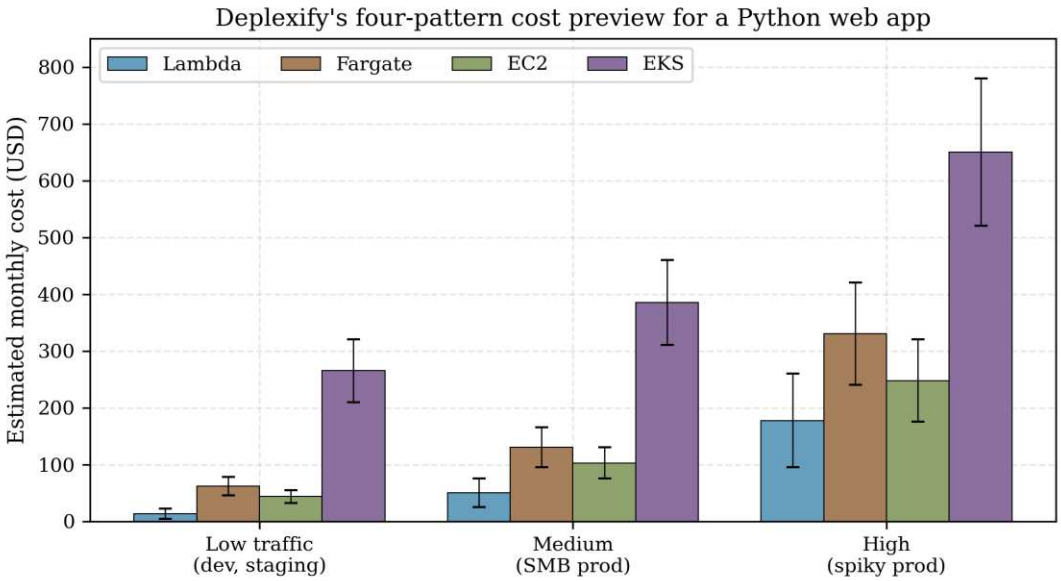


Fig. 10. *Deplexify's four-pattern cost preview for a representative Python web application at three traffic levels.*

F. Feature coverage

Fig. 9 is a radar plot on eight feature dimensions: bring-your-own-cloud, arbitrary AWS service support, IaC inspectability, multi-pattern cost preview, human approval gates, pull-request-based refactor delivery, security scanning, and automated teardown. The pure PaaS products (Vercel, Heroku) score low on every dimension because they hide the infrastructure by design. AWS App Runner scores modestly higher on BYOC and arbitrary services but has no cost preview or human gates. Pulumi AI, the closest analogue in IaC-generation space, scores well on BYOC, inspectability, and arbitrary services but low on cost preview and human gates. Deplexify covers all eight dimensions substantially. This is not a claim of superiority; it is a claim of scope. Deplexify chooses to do more than a PaaS, and the trade-off is complexity and infrastructure ownership.

TABLE IV

Deplexify against the closest Platform-as-a-Service and Infrastructure-as-Code alternatives.

System	Runs in user's account	IaC inspectable	Cost preview before commit	Human approval gates	Arbitrary AWS services
Vercel	No	No	No	No	No
Railway	No	No	Limited	No	No
Render	No	No	Limited	No	No
AWS App Runner	Yes	Partial	Limited	No	Limited
Pulumi AI	Yes	Yes	Partial	Partial	Yes
Deplexify	**Yes**	**Yes**	**Yes (4 patterns)**	**Yes (3)**	**Yes**

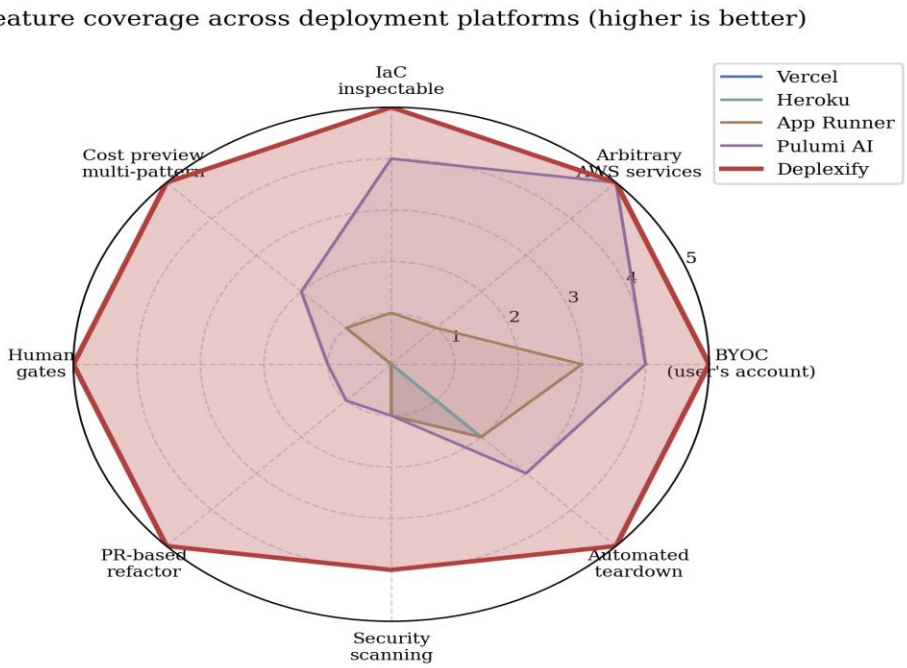


Fig. 9. *Radar plot of feature coverage across five deployment platforms on eight dimensions.*

G. Analytical model of grounding on hallucination

Fig. 11 plots the analytical bound on hallucination probability from equation (4). At grounding fraction $\alpha = 0$, hallucination probability in the output equals hallucination probability in the model's judgment. As α approaches 1 the output hallucination bound approaches 0 regardless of model quality. Deplexify's analysis stage operates at approximately $\alpha = 0.85$: the AnalysisResult contains six fields (framework, runtime, ports, database, healthcheck path, secrets summary) of which four are populated by scanner data (packages via Syft, secrets via Gitleaks, port hints via regex, database hints via package presence) and two by the language model (framework name, deploy-readiness explanation). Under a typical model with Pr(J-hallucination) = 0.40, the bound gives approximately 6 percent hallucination probability in the analysis output. The three curves in Fig. 11 show the same bound for three model qualities. The point of the analytical model is that grounding is a stronger reliability lever than model quality: a well-grounded system with a mediocre model outperforms a weakly-grounded system with a strong model.

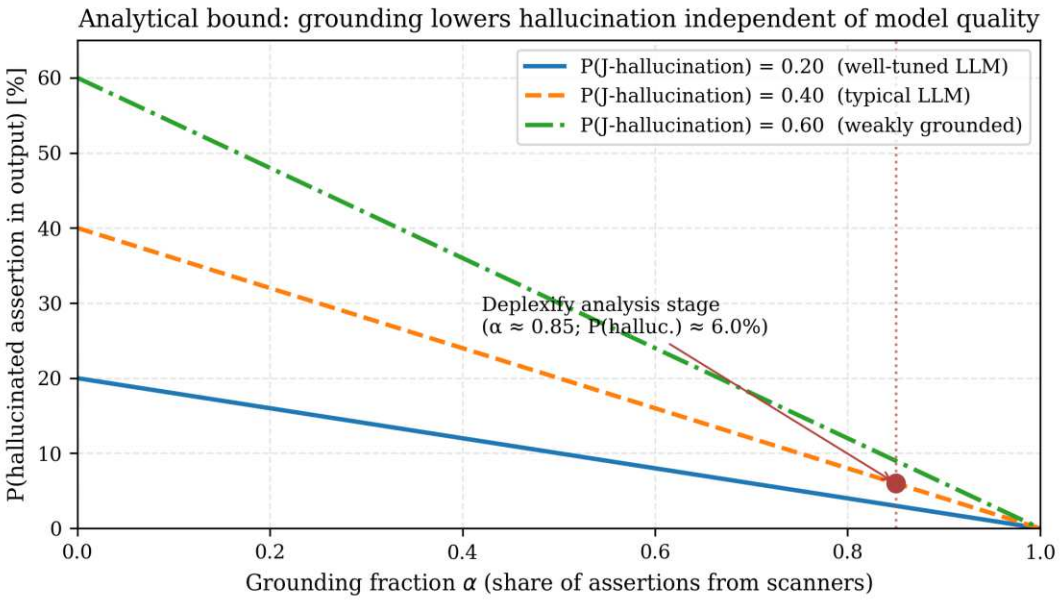


Fig. 11. *Analytical bound from equation (4). Grounding fraction α on the horizontal axis; probability of a hallucinated assertion in the AnalysisResult on the vertical axis; three curves for three model qualities.*

VIII. DISCUSSION, LIMITATIONS, AND FUTURE WORK

A. Discussion

Three observations from the implementation are worth recording.

The first concerns the value of grounding. The Claude analysis agent is explicitly instructed not to re-derive scanner facts. In an early prototype the same scanner output was provided but the instruction was absent. The agent re-emitted package lists, sometimes correctly and sometimes hallucinated. Adding the instruction and pairing it with a structured output schema eliminated re-derivation. This is consistent with the analytical bound in equation (4) and with the empirical literature on grounding [8] and package hallucinations [4].

The second concerns the cost of constraint. Restricting the analysis worker's tools to Read, Glob, and Grep makes the agent measurably safer: an agent that cannot mutate the workspace cannot escape it. The cost is a slight slowdown in cases where the agent would have benefited from executing a small command such as resolving a Python import. The slowdown is acceptable because correctness matters more than speed at this stage. When the refactoring agent runs, the tradeoff inverts: write access is required, and the safety story moves to the layered planner-filter, applier-scoping, and post-hoc workflow scrub described in Section IV-B.

The third concerns the placement of human gates. The three gates were chosen because they corresponded to decisions the team would not have made on a colleague's behalf without asking. The trust-calibration literature [22], [23] and the cost-benefit inequality of equation (5) arrive at the same conclusion through a different route.

B. Limitations

Several limitations are recorded honestly. Concurrency in the current build relies on threading and a shared volume rather than a distributed job queue, which bounds horizontal scale to a single node.

Terraform state lives on a Docker volume rather than in S3 with DynamoDB locking; concurrent runs against the same project are unsafe. JWTs are stored in localStorage rather than an httpOnly cookie; this is a known XSS exfiltration risk. A single Fernet key encrypts all tenant credentials; per-tenant envelope encryption is planned but not implemented. The rate-limit store is in-memory and resets on process restart. The worker has unrestricted network egress while holding customer credentials. All measurements in Section VI come from a single production VM in a single region. The three approval gates are argued from the literature, the case study, and equation (5); no user study has been conducted, and a user study is future work. The comparative analysis in Section VII relies on published benchmarks and vendor documentation for competitor systems rather than head-to-head re-benchmarking on identical hardware; this is standard practice in academic comparison but is a source of measurement noise that should be acknowledged.

C. Future work

The most interesting line of future work is what we call the Day-2 Agent: an autonomous component that retains read access to both the codebase and the cloud account and opens pull requests to remediate CVEs, AWS API deprecations, configuration drift, cost regressions, and performance anomalies over the lifetime of the deployment. This shifts the system from a one-shot deployment transaction to a continuous agent-mediated maintenance relationship. Other tracked work includes multi-cloud generalization to GCP and Azure; adding policy-as-code evaluation with Open Policy Agent or Sentinel on the generated plan so that constraint operates on outputs as well as on inputs; the socket proxy or rootless-build migration; remote Terraform state with locking; a durable queue for distributed orchestration; and a controlled user study to validate the gate-placement inequality against measured human performance.

IX. CONCLUSION

Deplexify is a working agentic system that transforms a GitHub repository URL into a live AWS deployment with bounded human supervision. Two design choices carry most of the system's reliability: grounding the language model in deterministic scanner output, formalized by the hallucination bound of equation (4), and constraining LLM-influenced infrastructure code through an allow-list envelope so that model influence on Terraform cannot exceed a verified boundary. A six-stage pipeline plus a monitoring plane, with three approval gates placed according to the cost-benefit inequality of equation (5), keeps the human at the decisions where automated error is most expensive. Measured latencies met the proposal targets; twelve of thirteen adversarial probes were blocked at review time and the thirteenth was fixed during the review. Comparative analysis against seven public deployment platforms and six published IaC generation baselines positions the system as a wider-scope but slower alternative to Platform-as-a-Service products, and as a categorically different approach to IaC accuracy than pure LLM generation. The approach is not a novel mechanism. It is a careful implementation of the consensus the literature describes, evaluated against the codebase it documents.

ACKNOWLEDGMENT

The authors thank Dr. Ihtisham Ali for academic supervision throughout the project and the School of Computing Sciences at PAF-IAST for the coursework that grounded the engineering decisions in this paper.

FUNDING DECLARATION

This work hasn’t received any funding grant.

CLINICAL TRIAL NUMBER

Clinical trial number “Not Applicable”

ETHICS, CONSENT TO PARTICIPATE, AND CONSENT TO PUBLISH

Not Applicable.

DATA AVAILABILITY STATEMENT

The data supporting the findings of this study are available from the corresponding author on reasonable request. The evaluation measurements and results generated during this study are reported in the manuscript. No separate publicly archived dataset was used or generated for this study.

REFERENCES

[1] Gartner, "Gartner Forecasts Worldwide Public Cloud End-User Spending to Total $723 Billion in 2025," press release, Nov. 19, 2024.

[2] Flexera, 2025 State of the Cloud Report. Itasca, IL, USA: Flexera, Mar. 2025.

[3] Flexera, 2026 State of the Cloud Report. Itasca, IL, USA: Flexera, 2026.

[4] J. Spracklen, R. Wijewickrama, A. H. M. N. Sakib, A. Maiti, B. Viswanath, and M. Jadliwala, "We Have a Package for You! A Comprehensive Analysis of Package Hallucinations by Code Generating LLMs," in Proc. 34th USENIX Security Symp., 2025.

[5] DORA / Google Cloud, 2024 Accelerate State of DevOps Report, 2024.

[6] R. Pujar et al., "A Survey of Using Large Language Models for Generating Infrastructure as Code," arXiv preprint arXiv:2404.00227, 2024.

[7] P. T. Kon, J. Liu, Y. Qiu, W. Fan, T. He, L. Lin, H. Zhang, O. M. Park, G. S. Elengikal, Y. Kang, A. Chen, M. Chowdhury, M. Lee, and X. Wang, "IaC-Eval: A Code Generation Benchmark for Cloud Infrastructure-as-Code Programs," in Adv. Neural Inf. Process. Syst. (NeurIPS), 2024.

[8] R. Nekrasov, S. Fossati, I. Kumara, D. A. Tamburri, and W.-J. van den Heuvel, "IaC Generation with LLMs: An Error Taxonomy and A Study on Configuration Knowledge Injection," arXiv preprint arXiv:2512.14792, 2025.

[9] P. Jana, S. Davidson, B. Bhasker, A. Kan, A. Deoras, and L. Callot, "TerraFormer: Automated Infrastructure-as-Code with LLMs Fine-Tuned via Policy-Guided Verifier Feedback," in Proc. 48th IEEE/ACM Int. Conf. Softw. Eng. (ICSE), Rio de Janeiro, Brazil, Apr. 2026.

[10] R. N. H. Khan, D. Wasif, J. Cho, and A. Butt, "Multi-Agent Code-Orchestrated Generation for Reliable Infrastructure-as-Code," arXiv preprint arXiv:2510.03902, 2025.

[11] T. Zhang, S. Pan, Z. Zhang, Z. Xing, and X. Sun, "Deployability-Centric Infrastructure-as-Code Generation: Fail, Learn, Refine, and Succeed through LLM-Empowered DevOps Simulation," in Proc. 34th ACM Int. Symp. Found. Softw. Eng. (FSE), 2026.

[12] S. Davidson, L. Sun, B. Bhasker, L. Callot, and A. Deoras, "Multi-IaC-Eval: Benchmarking Cloud Infrastructure as Code Across Multiple Formats," arXiv preprint arXiv:2509.05303, 2025.

[13] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao, "ReAct: Synergizing Reasoning and Acting in Language Models," in Proc. Int. Conf. Learn. Representations (ICLR), 2023.

[14] N. Shinn, F. Cassano, E. Berman, A. Gopinath, K. Narasimhan, and S. Yao, "Reflexion: Language Agents with Verbal Reinforcement Learning," in Adv. Neural Inf. Process. Syst. (NeurIPS), 2023.

[15] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press, "SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering," in Adv. Neural Inf. Process. Syst. (NeurIPS), 2024.

[16] C. S. Xia, Y. Deng, S. Dunn, and L. Zhang, "Agentless: Demystifying LLM-based Software Engineering Agents," arXiv preprint arXiv:2407.01489, 2024.

[17] Amazon Web Services, "Amazon Q Developer," documentation, 2025.

[18] Anthropic and contributors, "Model Context Protocol Specification," 2025. [Online]. Available: https://modelcontextprotocol.io

[19] X. Hou et al., "Model Context Protocol (MCP): Landscape, Security Threats, and Future Research Directions," arXiv preprint arXiv:2503.23278, 2025.

[20] M. M. Hasan, H. Li, E. Fallahzadeh, G. K. Rajbahadur, B. Adams, and A. E. Hassan, "Model Context Protocol (MCP) at First Glance: Studying the Security and Maintainability of MCP Servers," arXiv preprint arXiv:2506.13538, 2025.

[21] X. Li and X. Gao, "A First Look at the Security Issues in the Model Context Protocol Ecosystem," arXiv preprint arXiv:2510.16558, 2025.

[22] J. D. Lee and K. A. See, "Trust in Automation: Designing for Appropriate Reliance," Human Factors, vol. 46, no. 1, pp. 50-80, 2004.

[23] K. Lazaros, A. G. Vrahatis, and S. Kotsiantis, "Human-in-the-Loop Artificial Intelligence: A Systematic Review of Concepts, Methods, and Applications," Entropy, vol. 28, no. 4, art. no. 377, Mar. 2026, doi: 10.3390/e28040377.

[24] J. Ocasio and M. Mullarkey, "From Human-in-the-Loop to Human-on-the-Loop: The VALID Framework for Resilient Trust in Autonomous AI Agent Contract Execution," in Proc. Int. Conf. Design Sci. Res. Inf. Syst. Technol. (DESRIST), LNCS vol. 16605, 2026.

[25] Q. Cheng et al., "AI for IT Operations (AIOps) on Cloud Platforms: Reviews, Opportunities and Challenges," arXiv preprint arXiv:2304.04661, 2023.

[26] N. Forsgren, J. Humble, and G. Kim, Accelerate: The Science of Lean Software and DevOps. Portland, OR, USA: IT Revolution Press, 2018.

[27] E. O'Donoghue, B. Boles, C. Izurieta, and A. M. Reinhold, "Impacts of Software Bill of Materials (SBOM) Generation on Vulnerability Detection," in Proc. ACM Workshop Softw. Supply Chain Offensive Res. and Ecosystem Defenses (SCORED), 2024, doi: 10.1145/3689944.3696164.

[28] Open Policy Agent Community, "Open Policy Agent Documentation," 2025. [Online]. Available: https://www.openpolicyagent.org

[29] Aqua Security, "tfsec: Security Scanner for Terraform," 2024. [Online]. Available: https://github.com/aquasecurity/tfsec

[30] HashiCorp, "Terraform Documentation," 2025. [Online]. Available: https://developer.hashicorp.com/terraform/docs

[31] Vercel, "Vercel Documentation and Fluid Compute," 2025-2026. [Online]. Available: https://vercel.com/docs

[32] Railway, "Comparing top PaaS and deployment providers," Railway blog, Oct. 2025.

[33] Amazon Web Services, "AWS App Runner improves performance for image-based deployments," Dec. 2023.

[34] S. Ramirez, "FastAPI Documentation," 2025. [Online]. Available: https://fastapi.tiangolo.com

[35] PostgreSQL Global Development Group, "PostgreSQL 16 Documentation," 2025.

[36] Vercel, "Next.js 14 Documentation," 2024-2025.

[37] M. Jones, J. Bradley, and N. Sakimura, "JSON Web Token (JWT)," IETF, RFC 7519, May 2015.

[38] D. Hardt, Ed., "The OAuth 2.0 Authorization Framework," IETF, RFC 6749, Oct. 2012.

[39] N. Peck, "Build your application easily and efficiently with serverless containers," AWS re:Invent 2022 talk / Nathan Peck blog, 2022.

[40] D. K. Dalai, "Taming Cold Starts on AWS Fargate: The Architecture Behind Sub-5-Second Task Launches," AWS in Plain English, Oct. 2025.

[41] The Software Scout, "Railway vs Render 2026: Pricing & Speed Tested," 2026.