CloudBot: Autonomous End-to-End Cloud Deployment from Code to Infrastructure

Oluwatobiloba Ódùnsī[1], Aravind Mohan[2], Seraj Al Mahmud Mostafa[1], Jianwu Wang[1]
[1]*Department of Information Systems, University of Maryland, Baltimore County,* Baltimore, MD, USA
[2]*Department of Computer Science, McMurry University,* Abilene, TX, USA

***Abstract*—While cloud platforms offer extensive services for running and scaling applications, automatically deploying code from GitHub repositories to cloud infrastructure remains a manual, error-prone process requiring specialized expertise. Current DevOps tools and Infrastructure-as-Code (IaC) frameworks rely on fixed templates and cannot adapt to diverse application requirements automatically. We propose CloudBot, a toolkit that automates the complete deployment workflow by integrating code analysis, IaC generation, and infrastructure provisioning into a unified pipeline. CloudBot employs a pipeline where specialized components, GitHub Analyst, Cloud Architect, and Cloud Engineer, work sequentially to extract requirements, design infrastructure, and generate validated Terraform templates. The toolkit uses large language models enhanced with retrieval-augmented generation to map application needs to infrastructure specifications. We evaluate CloudBot with AWS cloud across three use cases: text processing, image analysis, and video processing. CloudBot achieves consistent deployment success, with all configurations deploying and executing correctly. Comparing generated infrastructure against expert-written baselines reveals similarity scores of 18-43% across syntax, semantic, and functional dimensions. While generated configurations successfully deploy, they tend toward over-provisioning compared to minimal expert specifications. As the first end-to-end automated deployment system, CloudBot demonstrates LLM-based infrastructure automation feasibility while establishing quantitative baselines for measuring future improvements toward expert-level generation quality.**

***Index Terms*—Cloud Infrastructure Automation, Infrastructure as Code (IaC), Large Language Models (LLM), Retrieval-Augmented Generation (RAG), Multi-Agent, Cloud Computing, GitHub Code Analysis.**

I. Introduction

Cloud application deployment has emerged as a critical challenge for developers and organizations adopting cloud platforms. While cloud computing promises flexibility, scalability, and on-demand resources, moving an application from a GitHub repository to running infrastructure on AWS, Azure, or Google Cloud remains highly manual and error-prone [1]. Developers must understand application requirements, translate them into infrastructure specifications, configure services, provision resources, deploy code, and validate operations steps that demand specialized expertise and consume significant time. The challenge of automated "code-to-cloud" deployment is both timely and important, highlighting the gap between increasing application complexity and limited automation available to developers [2], [3].

Consider a data science team developing a video analysis application. To deploy this to the cloud, they must determine appropriate compute instances, identify system dependencies like `ffmpeg`, configure storage, set up security groups, write IaC templates, validate configurations, provision infrastructure, transfer code, install dependencies, execute the application, and collect results. Each step requires cloud platform knowledge, introduces potential errors, and consumes time better spent on application logic. When requirements change or the application moves to a different provider, much of this work must be repeated.

Infrastructure-as-Code platforms like Terraform and AWS CloudFormation have improved resource management through declarative specifications, but still demand substantial manual expertise [4], [5]. IaC frameworks provide consistency once templates are written, but do not automate the critical first step: understanding application requirements and translating them into infrastructure specifications. Developers must manually analyze code, identify dependencies, determine resource needs, and encode this knowledge into templates-precisely the expertise-intensive steps that limit cloud adoption.

Recent advances in large language models (LLMs) show promise in analyzing codebases [6], [7]. LLM-driven IaC generation has demonstrated syntactically correct specifications, yet significant gaps remain: deployment success rates as low as 30% without iterative refinement, poor intent alignment, and minimal security compliance [8], [9]. These systems assume infrastructure requirements are explicitly specified in natural language prompts rather than automatically extracted from code. The process remains fragmented, leaving optimization opportunities at the intersection of code analysis, infrastructure specification, and deployment execution untapped [10].

To address these limitations, we propose CloudBot, a toolkit that integrates code analysis, resource optimization, and IaC generation into a single coordinated pipeline. CloudBot leverages LLMs, retrieval-augmented generation (RAG), to bridge the gap between application repositories and operational infrastructure. Our research addresses three key questions:

RQ1: Can LLMs generate Infrastructure-as-Code that successfully deploys and executes applications end-to-end without manual intervention?

RQ2: How closely does automatically generated infrastructure match expert-written baseline configurations?

RQ3: What are the specific limitations of current LLM-based infrastructure generation, and what mechanisms bridge the gap toward expert-level quality?

CloudBot makes three key contributions addressing these questions. **First**, it automates the full workflow from reposi-

tory analysis to deployed application execution, establishing the first complete automation of the code-to-cloud cycle. **Second**, it employs a pipeline where the GitHub Analyst extracts requirements, the Cloud Architect craft the prompt, and the Cloud Engineer generates validated Terraform templates through RAG-enhanced generation and intelligent repair. **Third**, through systematic experiments across text processing, image analysis, and video processing applications, CloudBot establishes quantitative baselines measuring current LLM-based generation capabilities. Our results show consistent deployment success while revealing 18-43% similarity to expert baselines, indicating that current LLMs produce working but non-minimal infrastructure.

These contributions demonstrate both feasibility and current limitations of fully automated cloud deployment. CloudBot answers RQ1 affirmatively: LLMs can generate deployable IaC for complete automation. Our similarity analysis addresses RQ2, quantifying the quality gap. The patterns we identify address RQ3, revealing areas needing improvement: reducing over-provisioning, better capturing minimal requirements, and handling edge cases in dependency specifications.

The remainder of this paper is organized as follows. Section II reviews related work in LLM-driven IaC generation, RAG for cloud systems. Section III describes CloudBot's architecture, detailing the pipeline and workflow. Section IV presents our experimental methodology. Section V reports results demonstrating deployment success and quantifying similarity to expert baselines. Section VI discusses findings, implications, and limitations. Section VII concludes with contributions and future directions.

II. Related Work

A. LLM-Driven Infrastructure-as-Code Generation

Large language models have shown promise in automating IaC generation from natural language descriptions. WISDOM-Ansible [11] fine-tuned transformer models on Ansible YAML data, achieving a BLEU score of 66.67, outperforming Codex-Davinci-002 (50.4). However, these approaches focus primarily on syntactic correctness without validating actual deployability.

IaCGen [8] addressed this gap through iterative feedback mechanisms, improving Claude-3.5's deployment success from 30.2% to 98% over 25 iterations. Despite these improvements, significant challenges remain: IaCGen achieved only 25.2% accuracy in intent alignment and 8.4% in security compliance, indicating that LLMs struggle with application-specific requirements and security best practices. The IaC-Eval benchmark [9] revealed even starker limitations, with GPT-4 achieving only 19.36% pass@1 accuracy on 458 AWS scenarios, compared to 86.6% on Python benchmarks. Traditional enhancement strategies like few-shot prompting degraded performance, while RAG provided only modest improvements (6.14% on average).

Beyond generation accuracy, Vo et al. [12] found that 84.5% of Terraform repositories contain code smells, with LLMs detecting 92.9% compared to traditional linters' 11.9-14.3%. A broader survey [13] documented these challenges, noting the scarcity of IaC training data and difficulty evaluating correctness without deployment. These systems share a critical limitation: they focus exclusively on IaC generation, assuming infrastructure requirements are pre-specified in natural language. They do not address automatic extraction of requirements from application code, nor handle post-deployment execution and result collection.

B. Retrieval-Augmented Generation for Cloud Systems

Retrieval-augmented generation (RAG) [14] enhances LLM-based systems by grounding them in domain-specific knowledge. Yang et al. [15] showed that agents with retrieved service documentation make better VM provisioning decisions by accessing current cloud service information and pricing models. SARGE [16] applies RAG to detect IaC misconfigurations by retrieving security policies during analysis, identifying violations invisible to syntax checking alone.

Choudhary et al. [17] demonstrated RAG's effectiveness in warehouse robotics automation, where LLMs access dynamic inventory and operational data to guide robot task allocation. Their system improved query accuracy from 75% to 100% and eliminated hallucinations by retrieving real-time warehouse state, showing RAG's potential for infrastructure automation requiring dynamic context. However, existing RAG systems for cloud automation operate in isolation on specific tasks like provisioning or security scanning. They lack integration across the full deployment pipeline and use static retrieval strategies rather than adapting to evolving workflow context.

C. Multi-Agent Systems for Cloud Operations

Multi-agent LLM systems have shown promise for complex tasks requiring diverse expertise. Recent surveys [18], [19] highlight that multi-agent architectures outperform single-agent systems by distributing responsibilities among specialized agents. In software engineering, agents take on distinct roles such as planner, implementer, and reviewer, collaborating to produce higher-quality outputs [20].

RCAgent [21] uses multiple agents for root cause analysis in cloud environments, collecting system data and diagnosing issues collaboratively. However, it focuses on debugging existing deployments rather than automating initial provisioning. General-purpose frameworks like AutoGen [22] and CrewAI provide infrastructure for building collaborative systems but require significant customization for cloud deployment tasks. They lack pre-configured agents specialized for infrastructure automation or built-in knowledge about cloud services and deployment patterns.

D. How CloudBot Differs

Unlike existing work that focuses on isolated stages, CloudBot provides true end-to-end automation from repository analysis to deployed application. While IaCGen and similar systems assume infrastructure requirements are specified in natural language prompts, CloudBot automatically extracts these requirements by analyzing application code, dependencies, and

structure. CloudBot assigns specialized roles (GitHub Analyst, Cloud Architect, Cloud Engineer) that work sequentially. Most critically, existing systems stop at infrastructure provisioning, but CloudBot completes the loop by executing applications, and collecting outputs. By combining automated requirement extraction, RAG-enhanced generation with validation and repair, and full execution management, CloudBot addresses the fragmentation in prior work and delivers a cohesive solution for practical cloud deployment automation.

III. System Architecture and Design

A. Overview

CloudBot is designed as an end-to-end automated pipeline that transforms application source code into fully deployed cloud infrastructure, as depicted in Figure 1. Given a repository URL as input, the system performs a sequence of operations: it analyzes the repository structure and requirements, generates Infrastructure-as-Code (IaC) specifications using large language models, validates and repairs the generated code, deploys the infrastructure on a cloud platform, and finally executes the application while collecting performance metrics and outputs.

The system architecture follows a modular design philosophy where each component performs a well-defined task and communicates through standardized interfaces. This separation of concerns enables independent development, testing, and replacement of individual components while maintaining overall system integrity. The complete workflow can be formally represented as a function composition:

$$\text{CloudBot}(R) = D \circ E \circ V \circ G \circ A(R), \tag{1}$$

where R represents the input repository, A is the analysis function, G is the generation function, V is the validation function, E is the extraction function, and D is the deployment function. Each function transforms its input into a form suitable for the next stage in the pipeline.

B. GitHub Analyst

The GitHub Analyst is the first component of the toolkit. Its main role is to collect and prepare code from a GitHub repository for further processing. It takes a GitHub project URL as input, verifies it, and then downloads the project files into a clean, organized folder. The files are saved with a timestamp to ensure version tracking, and any Git history is removed to keep the snapshot simple and consistent. This step ensures that the next component the Cloud Architect receives a stable, ready-to-use version of the repository. In essence, the GitHub Analyst automates the process of retrieving and organizing source code so the rest of the system can focus on analyzing and generating the required infrastructure setup.

C. Cloud Architect

The Cloud Architect is the second component of the toolkit and acts as the bridge between the raw code and the infrastructure generation process. Its primary role is to interpret the project context from the cloned repository and prepare structured input for the next stage of the pipeline. It processes the collected repository, extracts relevant information about the project's purpose or functionality, and transforms that understanding into a clear and organized prompt that guides the Cloud Engineer. In simpler terms, the Cloud Architect translates what the project is about into the infrastructure that needs to be built. This ensures that the system can automatically and accurately generate the Infrastructure as Code (IaC) needed to deploy the application in the cloud.

The output of this analysis is a structured prompt P that encapsulates the repository's infrastructure requirements:

$$P = f_{\text{parse}}(\text{README}, S, D, E), \tag{2}$$

where S represents the repository structure, D represents the dependencies, and E represents the identified entry point. The prompt is designed to be concise yet complete, explicitly requesting minimal but functional Terraform code that includes provider configuration, compute instance specifications, security group rules, and initialization scripts.

D. Cloud Engineer

Once the prompt is constructed, it is sent to the RAG-enhanced LLM service for infrastructure code generation. This component combines the power of large language models with retrieval-augmented generation to produce contextually appropriate Terraform code. The RAG system maintains a knowledge base of cloud provider documentation, common infrastructure patterns, and verified deployment templates.

The generation process can be modeled as:

$$T_{\text{raw}} = \text{LLM}(P, \text{RAG}(P, K)), \tag{3}$$

where T_{raw} is the raw Terraform code generated by the model, K represents the RAG knowledge base, and $\text{RAG}(P, K)$ retrieves relevant context from the knowledge base based on the prompt. The LLM service runs locally using Ollama [23], ensuring that no repository information or credentials are transmitted to external services. This design choice prioritizes security and privacy while maintaining the benefits of advanced language models.

The Cloud Engineer serves as the execution layer of the toolkit, integrating LLaMA 3.2 3B [24] with and a Retrieval-Augmented Generation (RAG) framework to produce accurate and deployable Infrastructure as Code (IaC). It operates within the Ollama environment, which locally manages both the LLaMA model and the mxbai-embed-large embedding model [25] used for the RAG. The RAG subsystem constructs a local knowledge base by processing the cloud provider documentation, cleaning and segmenting them, and transforming the text into vector representations using mxbai-embed-large. When a query or structured prompt is received, it is reformulated for clarity, embedded, and compared against the stored vectors through cosine similarity to identify the most relevant context. This context is passed to the LLaMA model, ensuring that code generation is grounded in precise and semantically aligned

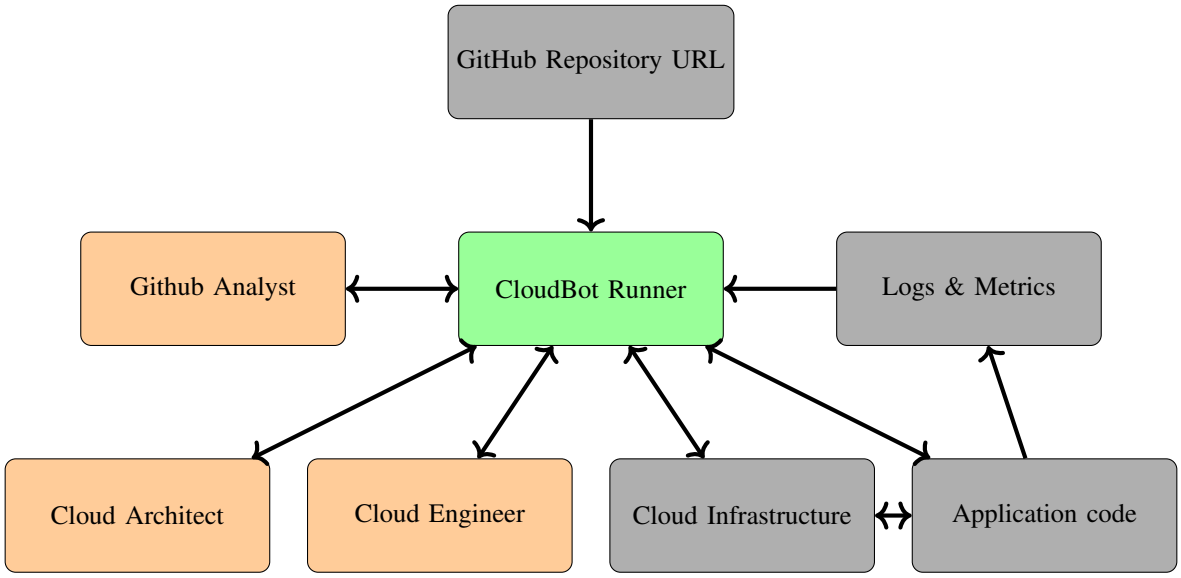

Fig. 1. CloudBot system architecture showing the CloudBot Runner as the central hub connecting the GitHub repository URL with the other components (GitHub Analyst, Cloud Architect and Cloud Engineer in orange), infrastructure provisioning, Application Code, and Logs & Metrics. Bidirectional arrows indicate information flow throughout the deployment pipeline.

knowledge. After generation, the Cloud Engineer performs an internal preparation step to ensure the produced IaC is ready for deployment. It isolates the generated code from any additional text, validates it against a verified baseline, and automatically corrects inconsistencies or omissions. This refinement process addresses the inherent variability of LLM outputs, guaranteeing that the final Terraform configuration is both syntactically correct and functionally reliable. Through the integration of retrieval, generation, and automated validation, the Cloud Engineer establishes a unified, locally managed pipeline capable of delivering reproducible, deployment-ready infrastructure code.

E. CloudBot Runner

The CloudBot Runner functions as the automation and orchestration layer of the entire system. Implemented as a GitHub Actions workflow, it serves as a continuous integration and continuous deployment (CI/CD) pipeline that manages the execution of all components beginning from repository acquisition to infrastructure deployment and application execution. Once triggered, it receives the GitHub repository URL and initializes the sequence by invoking the GitHub Analyst to clone and prepare the repository. It then calls the Cloud Architect, which interprets the project context and formulates structured prompts, followed by the Cloud Engineer, which retrieves contextual data through the RAG subsystem and generates the corresponding Terraform Infrastructure as Code (IaC) using the LLaMA model hosted locally via Ollama. After the IaC is produced, the CloudBot Runner proceeds to validate and deploy it within an AWS environment. It configures the necessary credentials, initializes Terraform, plans, and applies the infrastructure setup provisioning virtual resources such as EC2 instances as required. Once the infrastructure is successfully deployed, the workflow transitions into executing the application use case derived from the cloned repository, effectively running the Application code on top of the newly provisioned cloud infrastructure. The results from this execution, including logs and experimental outputs, are then written back into the cloned GitHub repository usecase in the CloudBot repository, ensuring versioned tracking of both the IaC and the experimental outcomes. Through this fully automated process, the CloudBot Runner transforms the entire pipeline from code acquisition to deployment and evaluation into a seamless workflow. It ensures reproducibility, reduces human intervention, and maintains operational consistency by managing the lifecycle of the toolkit within a unified, automated CI/CD framework.

IV. Experimental Setup

A. Experimental Objectives

Our experiments evaluate CloudBot's ability to automatically deploy applications from repository to running infrastructure across different complexity levels. We assess three key aspects: the accuracy of generated infrastructure specifications, the success rate of actual deployments, and the efficiency of the end-to-end pipeline. By comparing CloudBot against handcrafted baselines and measuring the contribution of individual components, we quantify the system's practical effectiveness for automated cloud deployment.

B. Use Case Selection

We selected three representative applications that progressively increase in complexity while covering diverse computational requirements and dependency patterns.

Word Count serves as our baseline case, processing plain text files to generate token frequency statistics in JSON

format. This Python application has minimal dependencies and represents simple data processing workloads common in cloud environments.

Image Analysis introduces moderate complexity, requiring computer vision libraries (OpenCV, Pillow) to extract features and generate classifications from image files. This case tests CloudBot's ability to identify and provision specialized dependencies beyond standard Python packages.

Video Analysis represents our most complex scenario, processing video clips to generate frame-level statistics using multimedia tools (ffmpeg, OpenCV). This case requires both Python libraries and system-level dependencies, challenging CloudBot to handle multi-layered dependency resolution and higher computational requirements.

C. Experimental Pipeline (Image Analysis Use Case)

A researcher supplies the URL of a GitHub repository containing an image-analysis program but lacks local compute. The **CloudBot Runner** receives the URL, invokes the **GitHub Analyst** to clone and stage the project, and calls the **Cloud Architect** to produce a structured prompt. The prompt concatenates the project goal, entry command, runtime needs, expected inputs/outputs, and AWS constraints extracted from the repo. The **Cloud Engineer** (running on EC2; **LLaMA + RAG** via **Ollama**) embeds the knowledge base and AWS best-practice references, retrieves context by **cosine similarity**, and generates Terraform IaC in a single shot. The output is cleaned, validated, and, if necessary, repaired against a ground-truth baseline.

The Runner then initializes Terraform and deploys the infrastructure described by the IaC into the **AWS sandbox** for this use case, an EC2 instance suitable for the image-analysis workload. It copies the application code from the cloned repository to the instance and executes the analysis. On completion, all artifacts prompts, retrieved context identifiers, IaC (generated and validated), infrastructure logs, application logs, metrics, timings, and instance metadata are written back to the repository.

Figure 2 illustrates the complete experimental workflow, showing how each test execution flows through six distinct stages in a cyclic manner.

Stage 1: Repository Trigger. Each experimental run begins when a repository URL is provided to the system, either through automated triggers or manual invocation. This initiates the complete deployment pipeline for the specified use case.

Stage 2: GitHub Actions/Local Runner (CloudBot Runner). The orchestration layer executes the CloudBot pipeline using either GitHub Actions for automated testing or a local workstation for development experiments. This layer maintains AWS credentials, manages workflow execution, and coordinates the subsequent stages.

Stage 3: CloudBot Agent (Components).

- **GitHub Analyst:** clones the repository, creates a clean, time-stamped workspace, and removes VCS state.
- **Cloud Architect:** inspects the staged repo and produces a structured deployment prompt. The prompt is assembled by concatenating fields derived from the repo (e.g., project goal, primary entry point/command, runtime requirements, expected inputs/outputs, and cloud constraints). To support this, repositories are expected to expose these elements in a consistent layout (e.g., top-level README with a "Project Goal" statement plus basic run instructions).

Input: Repository README

```
Project Goal: Run a simple word count program in the
      cloud
```

Output: Modified prompt

```
Please provide the terraform IaC to Run a simple
     word count program in the cloud in AWS EC2
```

- **Cloud Engineer:** runs in a prepared **EC2 host** (Ubuntu 24.04, **c5a.2xlarge**) as the generation environment. Within **Ollama**, it uses **LLaMA** for generation and **mxbai-embed-large** for the RAG. Project materials and reference documents are cleaned, segmented, embedded into **vector representations**, and matched to the prompt via **cosine similarity**. LLaMA performs single-shot Terraform IaC generation conditioned on the retrieved context. A built-in preparation step isolates code from prose, validates against a verified baseline, and repairs omissions or inconsistencies; if divergence remains, the baseline IaC is used.

RAG-Enhanced Generation
System Message:

```
You are a helpful assistant expert at extracting
     useful information from text. Bring in extra
     relevant information from outside the given
     context.
```

Enhanced Prompt: Original query + retrieved context

```
Please provide terraform IaC to Run a simple word
     count program in the cloud in AWS EC2

Relevant Context:
EC2 instances with user_data scripts, security
     groups for SSH (port 22), t2.micro instances,
     Amazon Linux 2 AMIs: ami-0c94855ba95c71c99, key
     pairs for SSH access.

Example pattern:
resource "aws_instance" "app_server" {
  ami = "ami-xxxxxxxx"
  instance_type = "t2.micro"
  key_name = "my-key"
  vpc_security_group_ids = [aws_security_group.app.id
       ]
  user_data = <<-EOF
    #!/bin/bash
    # Bootstrap script
  EOF
}
```

Output: Generated Terraform configuration

```
Terraform configuration for word count program:

```terraform
resource "aws_instance" "example_instance" {
  ami = "ami-0c94855ba95c71c99"
  instance_type = "t3.micro"
  vpc_security_group_ids = [aws_security_group.
       example.id]
  key_name = "example_key"
  user_data = <<-EOF
  #!/bin/bash
  echo 'name=My EC2 Instance'
  EOF
}
```
```

```
resource "aws_security_group" "example" {
  name = "my_ec2_instance"
  description = "Allow SSH access"
  vpc_id = aws_vpc.example.id

  ingress {
    from_port = 22
    to_port = 22
    protocol = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
}

resource "aws_vpc" "example" {
  cidr_block = "10.0.0.0/16"
}
```

Code Extraction Process: Extract fenced blocks, apply sanitization

Output: Sanitized Terraform file

```
resource "aws_instance" "example_instance" {
  ami = "ami-0c94855ba95c71c99"
  instance_type = "t3.micro"
  vpc_security_group_ids = [aws_security_group.
      example.id]
  key_name = "example_key"
  user_data = <<-EOF
  #!/bin/bash
  echo 'name=My EC2 Instance'
  EOF
}

resource "aws_security_group" "example" {
  name = "my_ec2_instance"
  description = "Allow SSH access"
  vpc_id = aws_vpc.example.id

  ingress {
    from_port = 22
    to_port = 22
    protocol = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
}

resource "aws_vpc" "example" {
  cidr_block = "10.0.0.0/16"
}
```

Configuration Validation

Output: Validated Terraform configuration

```
provider "aws" {
  region = "us-east-1"
}

resource "aws_instance" "my_ec2_instance" {
  ami = "ami-08982f1c5bf93d976"
  instance_type = "t2.micro"
  key_name = "cloudbot-east1"
  subnet_id = "subnet-017fca6bfc495ca34"
  vpc_security_group_ids = ["sg-012239f5f2ffabe18"]
  associate_public_ip_address = true
  tags = { Name = "cloudbot" }
}
```

Stage 4: Terraform Runner. The validated IaC undergoes deployment through the standard Terraform workflow (`init`, `plan`, `apply`). All Terraform operations are logged to capture provisioning details, timing information, and any errors encountered.

Stage 5: AWS Sandbox. The application executes on the **infrastructure defined by the generated IaC**. The specific resources vary by use case. For this image-analysis experiment, the workload runs on an EC2 instance; for other use cases, the IaC may provision different services. Each run uses a clean environment, network access is restricted (e.g., SSH only), and resources are destroyed on completion. The Runner transfers the cloned project to the target and executes the workflow on top of the deployed infrastructure.

Stage 6: Results Logging. The system collects and commits back to the repository: generated and validated IaC, Terraform logs, application outputs, timing measurements, and instance metadata. These artifacts feed subsequent iterations and comparative analysis.

D. Experimental Variants

To isolate the contribution of different CloudBot components and establish performance baselines, we compare three variants for each use case.

Baseline (Hand-Crafted IaC). An expert manually writes minimal Terraform code containing only the essential resources needed for each application. This represents the ideal target that CloudBot aims to match, establishing an upper bound on infrastructure minimality and serving as the ground truth for correctness evaluation.

Variant A (LLM-Only). We extract Terraform code from raw LLM output and validate syntax using `terraform validate`, but apply no semantic repairs. This variant is not actually deployed; we only measure syntactic correctness. Variant A quantifies the LLM's baseline generation capability and reveals common generation errors that require correction.

Variant B (CloudBot Complete). The full system executes with validation and repair mechanisms enabled. Generated IaC undergoes automated correction before deployment to actual AWS infrastructure. This variant measures CloudBot's end-to-end success rate, deployment time, and resource efficiency, demonstrating practical effectiveness.

Comparing these variants reveals: (1) the gap between raw LLM output and deployable infrastructure (Baseline vs. Variant A), (2) the value added by validation and repair (Variant A vs. Variant B), and (3) CloudBot's ability to approach expert-level infrastructure specifications (Variant B vs. Baseline).

E. Evaluation Metrics

We measure CloudBot's performance across multiple dimensions to provide a comprehensive assessment of system effectiveness.

Deployment Success Rate. The percentage of runs that successfully provision infrastructure, execute the application, and produce expected outputs without manual intervention. This is our primary metric for practical usability.

Infrastructure Correctness. We compare generated Terraform against baseline specifications, measuring: (1) presence of all required resources, (2) correctness of resource configurations, and (3) absence of unnecessary or redundant resources.

Syntax Validation Rate. The percentage of extracted IaC files that pass `terraform validate` before any repairs are applied, indicating raw LLM generation quality.

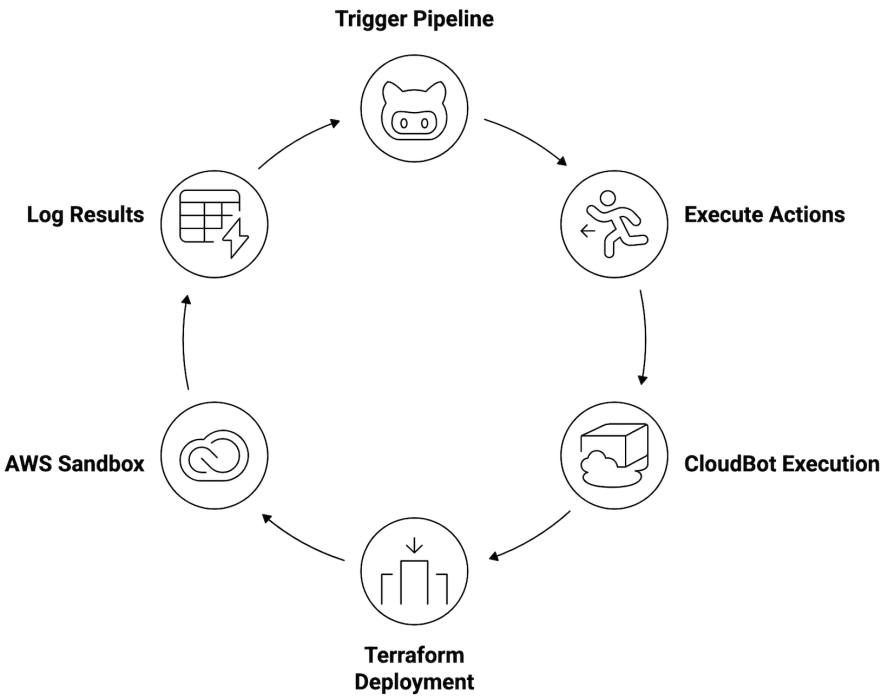

Fig. 2. Experimental pipeline showing the cyclic workflow from repository trigger through deployment and result logging. Each experiment flows through six stages: (1) repository trigger initiates the pipeline, (2) GitHub Actions or local runner executes the workflow, (3) CloudBot agents manage analysis and generation tasks, (4) Terraform runner deploys infrastructure, (5) AWS sandbox provides the test environment, and (6) results logging records outcomes before returning to the trigger for subsequent runs.

F. Experimental Configuration

All experiments use a standardized infrastructure configuration to ensure reproducibility and fair comparison. Target instances are AWS EC2 `c5a.2xlarge` running Ubuntu 24.04, deployed in a single fixed AWS region. The LLM Host runs on a separate EC2 instance serving models locally via Ollama with a RAG knowledge base containing AWS documentation and Terraform patterns. Each experimental run uses a fresh instance that is destroyed immediately after completion.

G. Data Collection and Reproducibility

For each experimental run, we systematically archive: repository commit hash, constructed prompts, `extracted.tf` and `validated.tf` files, complete Terraform logs (init, plan, apply, destroy), application outputs and error messages, timing data for each pipeline stage, and instance metadata (ID, IP, timestamps) with sensitive information redacted. Artifacts are organized by use case and run identifier, enabling both automated analysis and manual inspection. This comprehensive collection ensures experimental reproducibility and supports detailed investigation of edge cases and failure modes.

V. Experimental Results and Evaluation

We evaluate CloudBot's ability to generate deployable infrastructure code by comparing it against expert-crafted baselines across three use cases. Our evaluation addresses two key questions: (1) Can CloudBot generate IaC that successfully deploys and executes applications? (2) How closely does CloudBot's generated code match expert-written baselines in terms of syntax, semantics, and functionality?

A. Deployment Success

CloudBot successfully generated and deployed infrastructure across all three use cases over ten independent runs each (30 total deployments). Every generated Terraform configuration passed validation, provisioned AWS EC2 infrastructure, executed the target application, and produced expected outputs. This demonstrates CloudBot's ability to reliably automate the complete deployment pipeline from repository to running application without manual intervention.

TABLE I
AVERAGE SIMILARITY SCORES WITH STANDARD DEVIATIONS (TEN RUNS) COMPARING CLOUDBOT-GENERATED IAC TO EXPERT BASELINES.

Metric	UC1	UC2	UC3
Syntax	0.225 ± 0.052	0.295 ± 0.113	0.294 ± 0.083
Semantic	0.309 ± 0.099	0.431 ± 0.099	0.298 ± 0.089
Functional	0.183 ± 0.073	0.309 ± 0.131	0.191 ± 0.091

B. Similarity to Expert Baselines

While CloudBot consistently produces deployable infrastructure, we measure how closely its generated code matches minimal expert-crafted baselines using three complementary similarity metrics. Table I presents average similarity scores with standard deviations across ten runs for each use case. The standard deviations (ranging from 0.052 to 0.131) indicate moderate variability in generation quality across runs, reflecting the stochastic nature of LLM-based generation.

1) Syntax Similarity: Syntax similarity evaluates textual structure, measuring overlap in tokens, keywords, and code patterns. UC2 (Image Analysis) achieved the highest syntax similarity (0.295), with UC3 (Video Analysis) nearly identical (0.294). UC1 (Word Count) scored lowest (0.225), suggesting CloudBot generates code with different textual structure than the minimal expert baseline, despite functional correctness. The moderate scores (22-30%) indicate that while CloudBot produces valid Terraform syntax, the specific code organization and resource declarations differ substantially from expert implementations.

2) Semantic Similarity: Semantic similarity assesses conceptual overlap in infrastructure resources, examining whether configurations specify similar providers, resource types, variables, and architectural patterns. UC2 obtained the highest semantic similarity (0.431), indicating stronger alignment with expert-selected cloud services and configuration concepts. UC1 (0.309) and UC3 (0.298) showed moderate semantic alignment, suggesting CloudBot identifies appropriate infrastructure components but may select different resource configurations or include additional services compared to minimal baselines.

3) Functional Similarity: Functional similarity measures how alike the actual provisioned deployments are, using weighted features that emphasize critical components like resource types and provider configurations. UC2 again scored highest (0.309), while UC1 (0.183) and UC3 (0.191) scored substantially lower. These scores indicate that CloudBot's generated configurations, while deployable and functional, provision architecturally different infrastructure compared to expert baselines. The gap between semantic and functional similarity suggests that even when similar resources are specified, their actual configurations and relationships differ from expert designs.

C. Understanding the Similarity Gap

The moderate similarity scores (18-43%) despite consistent deployment success reveal an important characteristic of LLM-based IaC generation: current models can produce *working* infrastructure but do not yet match the minimality and precision of expert-crafted configurations. Analysis of generated code shows that CloudBot tends to include additional resources, more permissive security rules, and standard configurations rather than the tightly-scoped minimal specifications in expert baselines.

This gap represents both a limitation and an opportunity. The limitation is that automated generation does not yet achieve expert-level efficiency and minimality. The opportunity is that CloudBot establishes a working baseline (20-43% similarity with consistent deployment success) from which future improvements can be measured. Prior to CloudBot, no end-to-end system existed to automatically generate deployable IaC from repository analysis, making even this moderate similarity a meaningful step toward fully automated cloud deployment.

1) UC2 Performance Advantage: UC2 consistently outperformed UC1 and UC3 across all metrics. We attribute this to two factors: First, image processing applications with computer vision dependencies (OpenCV, Pillow) represent common deployment patterns well-represented in cloud documentation and IaC examples that inform the LLM's training data. Second, UC1's minimal requirements may have been over-provisioned by the LLM generating more resources than the tightly-scoped baseline, while UC3's multimedia dependencies (ffmpeg, opencv) introduced edge cases less common in typical cloud deployments. The variation across use cases highlights the influence of training data distribution on LLM generation quality.

D. Metric Computation and Aggregation

Each similarity family produces multiple sub-metrics per run. We report representative overall scores computed as follows:

- **Syntax Overall.** The score balances breadth and local structure:

$$\begin{aligned}\text{Syntax Overall} = {} & 0.50 * \text{token\_cosine} \\ & + 0.30 * \text{keyword\_Jaccard} \\ & + 0.20 * \text{3-gram\_Jaccard}.\end{aligned}$$

  Edit distance and char-level diff are retained only as diagnostics.
- **Semantic Overall.** The concept cosine over extracted Terraform concepts is used as the representative semantic score.
- **Functional Overall.** The weighted Jaccard over the feature set serves as the representative functional score, emphasizing resource types and providers.

Table II summarizes the representative scores and their components.

Each use case consists of multiple independent runs (ten in our experiments). For the tables in this section, we first compute per-run overall scores (one per family), then take the arithmetic mean across runs to obtain a single score per use case and family. This produces compact, comparable numbers

TABLE II
REPRESENTATIVE SCORE REPORTED PER FAMILY AND ITS COMPONENTS.

Family	Reported Overall (components)
Syntax	Weighted blend: token cosine (50%), keyword Jaccard (30%), 3-gram Jaccard (20%).
Semantic	Concept cosine over extracted Terraform concepts.
Functional	Weighted Jaccard over feature set (higher weights for resource types/providers).

while preserving the underlying detail in the supplemental report. Complete per-run metrics and detailed sub-scores are available in our repository for reproducibility.

E. Implications

These results demonstrate that CloudBot successfully automates end-to-end cloud deployment while generating infrastructure code that achieves moderate similarity (20-43%) to expert baselines. The consistent deployment success validates CloudBot's practical utility, while the similarity gap identifies clear directions for improvement: reducing over-provisioning, matching expert minimality, and better handling edge cases in dependency specifications. As the first system to provide fully automated deployment from repository to execution, CloudBot establishes both a working solution and quantitative baselines for measuring future progress in LLM-based infrastructure automation.

VI. DISCUSSION

A. Key Findings

CloudBot successfully automates end-to-end cloud deployment, achieving consistent success across all 30 experimental runs, thus answering RQ1 in Section 1 affirmatively. The moderate similarity scores (18-43%) compared to expert baselines quantify the quality gap (RQ2 in Section 1), revealing that current LLMs generate working but non-minimal infrastructure. UC2's superior performance across all metrics suggests that generation quality correlates with training data prevalence: applications with well-documented deployment patterns benefit from better LLM generation.

Analysis reveals consistent over-provisioning patterns where CloudBot includes standard configurations, more permissive security rules, and additional resources compared to minimal expert baselines. The validation and repair mechanism (RQ3 in Section 1) bridges the gap between raw generation and deployability, with the 0.6 similarity threshold empirically balancing correction effectiveness with preservation of valid content.

B. Practical Implications and Limitations

CloudBot provides immediate utility for research applications, prototyping, and educational environments by reducing deployment time from hours to minutes without requiring deep cloud expertise. The multi-agent architecture effectively decomposes complex deployment into specialized subtasks, mirroring expert workflows. However, production deployments requiring minimal resources or strict security still benefit from expert review.

Key limitations include targeting single-VM deployments on AWS only, three use cases that may not capture full application diversity, and reliance on a single LLM. Similarity metrics capture structural overlap but may not fully reflect operational quality like performance or cost efficiency. The RAG knowledge base, while comprehensive for AWS, may bias toward specific patterns.

C. Future Directions

Promising directions include improving generation minimality through fine-tuning on minimal IaC examples, extending to multi-service architectures and cross-cloud support, incorporating user feedback for iterative refinement, enhancing security through policy-based validation, and adding cost optimization objectives. CloudBot's modular architecture supports these extensions through its loosely-coupled components and expandable RAG knowledge base.

VII. CONCLUSION

CloudBot automates end-to-end cloud deployment from GitHub repositories to running infrastructure through an integrated pipeline combining code analysis, LLM-based IaC generation, validation, deployment, and execution. Our evaluation demonstrates consistent deployment success across text processing, image analysis, and video processing applications, with similarity scores of 18-43% to expert baselines indicating that current LLMs produce working but non-minimal infrastructure.

CloudBot makes three key contributions: providing the first complete end-to-end automation system, demonstrating effective multi-agent architecture for cloud deployment, and establishing quantitative baselines for measuring LLM-based infrastructure generation. Our results definitively answer the research questions: LLMs can generate deployable IaC end-to-end (RQ1), automated generation achieves 20-43% similarity to expert baselines (RQ2), and intelligent validation mechanisms bridge the generation gap with over-provisioning as the primary limitation (RQ3).

As LLMs continue improving and training data expands, the similarity gap will narrow. CloudBot's contribution lies in demonstrating current feasibility while establishing the framework and baselines for measuring future progress. The modular architecture, comprehensive evaluation methodology, and quantitative baselines provide a foundation for systematic advancement toward expert-level automated infrastructure generation, bringing seamless code-to-cloud deployment closer to reality.

REFERENCES

[1] V. Akuthota, R. Kasula, S. T. Sumona, M. Mohiuddin, M. T. Reza, and M. M. Rahman, "Vulnerability detection and monitoring using llm," in *2023 IEEE 9th International Women in Engineering (WIE) Conference on Electrical and Computer Engineering (WIECON-ECE)*. IEEE, 2023, pp. 309–314.

[2] L. PT, "Artificial Intelligence Challenges in software development: Risks, solutions, and future outlook — medium.com," https://medium.com/predict/artificial-intelligence-challenges-in-software-development-risks-solutions-and-future-outlook-1b2c7f524e78, [Accessed 06-10-2025].

[3] S. Morag, "Skills Shortage Threatens Cloud Security — forbes.com," https://www.forbes.com/councils/forbestechcouncil/2022/11/03/skills-shortage-threatens-cloud-security/, [Accessed 06-10-2025].

[4] E. Low, C. Cheh, and B. Chen, "Repairing infrastructure-as-code using large language models," in *2024 IEEE Secure Development Conference (SecDev)*. IEEE, 2024, pp. 20–27.

[5] "The Top 7 Challenges of Infrastructure as Code, And How to Solve Them — stackgen.com," https://stackgen.com/blog/7-challenges-infrastructure-as-code, [Accessed 12-07-2025].

[6] T. Sharma, M. Kechagia, S. Georgiou, R. Tiwari, I. Vats, H. Moazen, and F. Sarro, "A survey on machine learning techniques for source code analysis," *arXiv preprint arXiv:2110.09610*, 2021.

[7] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. D. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

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

[9] P. T. Kon, J. Liu, Y. Qiu, W. Fan, T. He, L. Lin, H. Zhang, O. M. Park, G. S. Elengikal, Y. Kang *et al.*, "Iac-eval: A code generation benchmark for cloud infrastructure-as-code programs," *Advances in Neural Information Processing Systems*, vol. 37, pp. 134 488–134 506, 2024.

[10] M. Shahin, M. A. Babar, and L. Zhu, "Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices," *IEEE access*, vol. 5, pp. 3909–3943, 2017.

[11] S. Pujar, L. Buratti, X. Guo, N. Dupuis, B. Lewis, S. Suneja, A. Sood, G. Nalawade, M. Jones, A. Morari *et al.*, "Automated code generation for information technology tasks in yaml through large language models," in *2023 60th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2023, pp. 1–4.

[12] Q.-H. Vo, H. Dao, and K. Fukuda, "Harnessing the power of llms for code smell detection in terraform infrastructure as code," in *2025 IEEE 49th Annual Computers, Software, and Applications Conference (COMPSAC)*. IEEE, 2025, pp. 533–542.

[13] K. G. Srivatsa, S. Mukhopadhyay, G. Katrapati, and M. Shrivastava, "A survey of using large language models for generating infrastructure as code," *arXiv preprint arXiv:2404.00227*, 2024.

[14] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel *et al.*, "Retrieval-augmented generation for knowledge-intensive nlp tasks," *Advances in neural information processing systems*, vol. 33, pp. 9459–9474, 2020.

[15] Z. Yang, A. Bhatnagar, Y. Qiu, T. Miao, P. T. J. Kon, Y. Xiao, Y. Huang, M. Casado, and A. Chen, "Cloud infrastructure management in the age of ai agents," *arXiv preprint arXiv:2506.12270*, 2025.

[16] M. K. Goyal and R. Chaturvedi, "Detecting cloud misconfigurations with rag and intelligent agents: A natural language understanding approach," *Available at SSRN 5271734*, 2025.

[17] K. Choudhary, S. Uruj, A. Pathak, V. Kalaichelvi, S. D. Shetty, R. Karthikeyan, T. Taha, and R. Muthusamy, "A retrieval-augmented generation (rag)-based llm for modern warehouse automation and management," in *2025 IEEE 21st International Conference on Automation Science and Engineering (CASE)*. IEEE, 2025, pp. 1688–1694.

[18] T. Guo, X. Chen, Y. Wang, R. Chang, S. Pei, N. V. Chawla, O. Wiest, and X. Zhang, "Large language model based multi-agents: A survey of progress and challenges," *arXiv preprint arXiv:2402.01680*, 2024.

[19] X. Li, S. Wang, S. Zeng, Y. Wu, and Y. Yang, "A survey on llm-based multi-agent systems: workflow, infrastructure, and challenges," *Vicinagearth*, vol. 1, no. 1, p. 9, 2024.

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

[21] Z. Wang, Z. Liu, Y. Zhang, A. Zhong, J. Wang, F. Yin, L. Fan, L. Wu, and Q. Wen, "Rcagent: Cloud root cause analysis by autonomous agents with tool-augmented large language models," in *Proceedings of the 33rd ACM International Conference on Information and Knowledge Management*, 2024, pp. 4966–4974.

[22] Q. Wu, G. Bansal, J. Zhang, Y. Wu, B. Li, E. Zhu, L. Jiang, X. Zhang, S. Zhang, J. Liu *et al.*, "Autogen: Enabling next-gen llm applications via multi-agent conversations," in *First Conference on Language Modeling*, 2024.

[23] "GitHub - ollama/ollama: Get up and running with OpenAI gpt-oss, DeepSeek-R1, Gemma 3 and other models. — github.com," https://github.com/ollama/ollama, [Accessed 11-10-2025].

[24] "llama3.2 — ollama.com," https://ollama.com/library/llama3.2, [Accessed 11-10-2025].

[25] "mxbai-embed-large — ollama.com," https://ollama.com/library/mxbai-embed-large, [Accessed 11-10-2025].