SvedbergOpen
DISSEMINATION OF KNOWLEDGE

International Journal of Artificial Intelligence and Machine Learning

Publisher's Home Page: https://www.svedbergopen.com/

Research Paper Open Access

Prompt-Based Secure Incremental State Management For Multi-Cloud Service Automation Using Terraform And Large Language Models

[1]Pali Deepak Phani Krishna, [2]*Dr. Sunitha Pachala

[1]Student, Department of CSE, Koneru Lakshmaiah Education Foundation, Green Fields, Vaddeswaram, Guntur Dist, Andhra Pradesh, India. E-mail: palideepak1549@gmail.com
[2]Associate Professor, Department of Computer Science and Engineering, Koneru Lakshmaiah Education Foundation, Green Fields, Vaddeswaram, Guntur Dist, Andhra Pradesh, India. E-mail: drsunitha.pachala@kluniversity.in

ABSTRACT
Prior work on prompt-driven multi-cloud provisioning demonstrated that Large Language Models (LLMs) can translate natural-language requests into Terraform configurations, dramatically reducing the time needed to provision infrastructure on AWS, Azure, and Google Cloud Platform (GCP). However, that system operated statelessly: it did not persist or reference a Terraform state file between sessions, so every prompt was treated as a fresh deployment. This caused destructive re-provisioning, in which existing resources were unnecessarily destroyed and recreated, resulting in downtime, potential data loss, and violation of the idempotency principle that Infrastructure-as-Code (IaC) systems are expected to uphold. This paper proposes a solution to that limitation: a secure, minimal state-referencing model in which only abstracted metadataresource type, resource count, and a non-identifying logical alias is persisted and exposed to the LLM, while sensitive identifying details such as names, IP addresses, ARNs, and endpoints remain solely inside an encrypted Terraform backend and are never transmitted to the language model. Using this abstracted summary alongside the user's prompt, the LLM is guided to generate targeted update operations matching, importing, or modifying existing resources instead of regenerating infrastructure from scratch. The proposed design is evaluated conceptually against the destructive-recreation baseline of the earlier system and demonstrated on a live AWS deployment sequence, showing improvements in idempotency, data-loss risk, security posture, and update latency, while keeping the system's original goal of simple, prompt-driven, non-expert-friendly multi-cloud management intact.

Keywords: Terraform, Infrastructure as Code, Large Language Models, State Management, Multi-Cloud, Idempotency, DevOps Automation, Secure Provisioning

1. Introduction

The rising popularity of multi-cloud strategies has caused the Infrastructure-as-Code (IaC) tools, like Terraform, to be the center of how organizations provisionand operate resources in AWS, Azure, and GCP. The previous research has been based on Terraform with a custom-trained LLM that allows users to create, query, and visualize cloud infrastructure in plain-English queries, reducing the time to fifteen to twenty minutes to less thanone minute to provision many services [1]. That system was developed to be based on three features: timely generation of code to be used in Terraform, tabular visualization of a live Terraform state file, and natural-language extraction of active resource properties, including IP addresses and endpoints.

Although it was useful in the initial provisioning, the previous system had a single structural weakness: it lacked a continuous record of what was already in place that could be referred to. Every new prompt was handled irrespective of previous deployments, and when a user requested the system to change or expand an existing service, the LLM did not have a good system to tell the difference between update this and create this out of nothing. This practicality implied that the resulting Terraform plan often destroyed the currently existing resource and replaced it, which was also explicitly defined as a drawback of that work, as well as the ensuing risks of data loss and non-idempotent operation [1].

A naive fix that merely continues and refeeds the entire terraform.tfstate file as input to the LLM on each prompt also creates a new issue: Terraform state files have sensitive and identifying infrastructure information in plaintext, such as private IP addresses, resource ARNs, database endpoints, and occasionally secrets. Trying to route this information via a prompt pipeline, in particular, a pipeline calling an external or shared LLM API, presents an unacceptable data-exposure surface. This paper deals with the two issues. It suggests a safe, minimal state-referencing model that only survives to de-identified, aggregate information resource type and count tagged with a locally generated logical alias, to allow the LLM to

reason about what already exists and produce incremental, update aware Terraform code, without ever being aware of the sensitive attributes that would render that infrastructure identifiable.
The rest of this paper is structured in the following way. Section 2 conducts a literature review on the related work on the code generationand automation of infrastructure based on LLM. Section 3 presents the problem as per the predecessor system. The proposed methodology is presented in section 4. Section 5 describes the system architecture. Details about implementation are discussed in section 6. In section 7, results and discussion are given. Section 8 is about the advantages of the suggested design, Section 9 is devoted to its limitations and future work directions, and the paper is concluded with Section 10.

2. Literature Survey

Ray et al. introduced PLBART, a sequence-to-sequence model, which is trained with denoising autoencoding on Java and Python code, paired with natural language and demonstrates good performance across code summarization, translation, and repair with minimal supervision [2].
Austin et al. tested program synthesis at scale (up to 137 billion parameters) on models of MBPP and MathQA-Python; program synthesis improved with scale, though none of the models could be consistently reliable to ensure correct program behavior without human inspection [3].
Brown et al. presented GPT-3, a 175-billion-parameter model that can achieve few-shot adaptation of tasks without task-specific fine-tuning, works well on translation, questionanswering, and text generation, but is still weak on tasks with no training examples that are related [4].
Chowdhery et al. have introduced PaLM, a 540-billion-parameter model that was trained on the Pathways system of Google and goes beyond previous state-of-the-art performance on multi-step reasoning tasks, as well as exhibits a high degree of code-related capability, but also raises concerns regarding bias and memorization at scale [5].
Dettmers et al. introduced QLoRA, a quantizationand fine-tuning technique that enables huge models to be fine-tuned on a single GPU, with 4-bit quantizationand low-rank adapters, reducing the resource barrier to fine-tuning to domain-specific models by a significant factor [6].
Feng et al. proposed a bimodal transformer called CodeBERT that was trained on both programming-language and natural-language data, which is effective at documentation generationand code search and demonstrates a good understanding of the code-language semantic alignment [7]. In addition to the encoder-only models, Nijkamp et al. also introduced CodeGen, an open, multi-turn program-synthesis LLM that is designed to generate code; its openness is a relevant concern to organizations which, like the previous system, would rather maintain a fine-tuned Terraform-generation model in-house instead of relying on a third-party API [20].
Chen et al. tested Codex, a GPT-based fine-tuned model on public GitHub Python code, on the HumanEval benchmark and reported good pass rates, but found longer and more logically complex programs to be challenging, and raised more general questions about the security implications of AI-generated code [8].
InfraCopilot by Masola suggested a conversational interface to edit IaC, an LLM-based intent parser with a graph-based architecture assembler, allowing infrastructure changes to be made by users without in-depth Terraform knowledge [9].
Pujar et al. have created Ansible Wisdom, a transformer-based system to create YAML automation scripts based on natural language, demonstrating that models trained on domain-specific configuration data can be better at this task than larger general-purpose models [10].
Zhang et al. suggested IaCGen, an iterative-feedback system to produce deployable IaC templates, and report a 98 percent success rate in deploying 153 test cases and the usefulness of validation loops as compared to single-shot code generation [11].
Outside the code-generation research community, state management has been a longstanding focus of the broader IaC and DevOps community, as the mechanism that draws the line between safe, incremental automationand destructive scripting. The state file, according to the official Terraform documentation of HashiCorp, is the single source of truth that identifies configuration to real-world resources and notes that state should be regarded as sensitive data, potentially containing plaintext credentials and attributes used to identify a resource; this is the motivation behind encrypted remote backends and very narrow access control instead of informal local storage [12]. A practitioner study of Terraform, by Brikman, also records that missing or poorly-managed state is among the most frequent sources of accidental resource destruction in teams based on IaC tooling, supporting the necessity of a state-referencing layer between IaC tooling and any assitive automation built on top of it [13]. In addition to this, industry best practices addressing secrets and infrastructure-metadata, including the suggestions of OWASP about data minimization, reinforce the overall idea that automation pipelines must not leak more than the minimum information they need to do a task, but instead convey complete infrastructure records through intermediate systems like an LLM API [14].
Further on, related to the exact multi-cloud automationenvironment discussed in this paper, Bishnoi et al. created a Terraform-driven framework of multi-cloud workload migration of containerized workloads

across multiple cloud providers, showing how complex it can be in reality to maintain multi-cloud deployments without an LLM in the loop [15]. Chanus and Aubertin investigated a tangible industrial deployment of the concept of pairing LLMs with IaC tooling, and they stated that the generation gains matched those of the prior system, but with a warning not to deploy the unverified changes to the production infrastructure [16]. Parthasarathy et al. suggested a multi-agent LLM system of autonomous CloudOps, which separates the tasks of the infrastructure into specialized agents, although, like the earlier system, there is no explicit mechanism by which agent behaviors are reconciled against state deployed previously [17]. The practitioner guide to multi-cloud architecture by Mulder describes the business logic behind organizations having workloads running on AWS, Azure, and GCP at the same time, and gives the deployment context within which the current paper is contextualized [18]. Here, the ISO/IEC 22123-1 cloud computing vocabulary standard is applied, on which multi-cloud terminology, namely, provider, resource, and workload is based, throughout the paper [19].

There are three current streams of work that further define the argument behind the security-conscious design, which is presented here and each is reexamined at the very point in the paper where it is used. Srivatsa et al. map the wider context of LLM-based IaC generationand pinpoint the issues of data exposure and ignorance of the already-provisioned infrastructure as gaping challenges that the current generation pipelines fail to address, which is why the state-referencing layer presented in Section 4 is proposed [21]. In a study by Kon et al. titled IaC-Eval, an infrastructure code generation benchmark of 458 human-curated Terraform scenarios, it was noted that even a powerful general-purpose model like GPT-4 only achieved 19.36 percent pass@1 accuracy on infrastructure code generation, in contrast to 86.6 percent on a similar Python benchmark; this finding was referred to in Section 8 as supporting the idea that the safe-apply checks of the proposed pipeline are an indispensable complement, not a substitute, to the state-referencing model [22]. Vargas et al. compared closed and open LLMs on AWS Terraform generation with the Checkov and Trivy policy scanners in a security-focused evaluation, and discovered that syntactic validity and security compliance are mostly independent properties - some models that consistently generate well-formed Terraform have failing security checks, and this observation is used in Section 4.5 and Section 8 to justify keeping sensitive state and policy enforcement out of the direct control of the LLM [23]. The closest to the reconciliation step presented in this paper, Mengistu et al. have introduced TerraRepair, which is a tool-grounded LLM agent that consults the installed provider schema and re-executes a scanner before giving a candidate fix, which can then be escalated to a human reviewer instead of creating a repair when deployment-specific context is unavailable; this pattern of escalate-rather-than-fabricate is also cited in Section 9 as a template to the proposed drift-detection extension in the paper [24]. Lastly, at the interface of the LLM itself, not the pipeline around it, Gim, Li, and Zhong suggest confidential prompting, where a confidential virtual machine and a secure partitioned-decoding protocol ensure that raw user prompts do not get seen by a cloud LLM provider but do not change the model's output; this hardware-based solution is contrasted in Section 4.5 with the data-minimization solution adopted in this paper and a blend of the two is discussed in Section 9 [25].

Combined, this body of work provides three directions of progress: (a) LLMs are becoming more competent at translating natural language into syntactically correct infrastructure code, albeit with lower accuracy than general-purpose code generation in this paper and in particular [21], [22]; (b) reliable, secure infrastructure automation requires accurate, security-conscious state tracking [12], [13], [14]; and (c) securing the inputs of an LLM pipeline can be achieved by either minimizing inputs, as in this paper, or hiding inputs cryptographically, as in confidential prompting, and in either case security conformance must be individually verified, not assumed [23], [24], [25]. The old system which succeeded the predecessor one had the former and lacked the latter [1]. The current work lies at the cross-section between these three lines and suggests a state-referencing mechanism which would be safe in itself to be exposed to an LLM-driven pipeline.

3. PROBLEM DEFINITION AND CONTRIBUTION OF THIS WORK

The former system created anentire Terraform plan each time a user ran the command and ran the default init-plan-apply lifecycle within an isolated workspace, without verifying the existence of a similar resource deployed in a previous session. Since Terraform is driven by configuration to determine what to create, change or destroy, and no record existed, the tool itself told Terraform that nothing existed. The consequence, recorded in the limitations of the predecessor work itself, was that every new prompt was like a new deployment cycle: the existing resources were rewrittenand recreated instead of being modified incrementally, stateful resources (like storage buckets and databases) were vulnerable to accidental data loss, and the system was unable to guarantee idempotency when a similar prompt was executed twice[1].

An immediate solution would be to continue to store the Terraform state file and inject it into the prompt context of the LLM with each request. This is an issue in two ways. First, state files can be huge and highly nested which makes them costly to pass in each LLM call. Second, and more significantly, state files have sensitive, identifying information: private IP addresses, instance identifiers, storage endpoints, ARNs, and,

in certain ill-configured scenarios, credentials. The transfer of this data into an LLM especially when the LLM can be served elsewhere introduces a security and privacy risk which was not present when the system was generating code solely by a prompt. The two-fold nature of the problem discussed in this paper is thus as follows: (i) how to provide the LLM with persistent context such that it will produce update-aware, non-destructive Terraform code, and (ii) how to do so without revealing sensitive identifiers of infrastructure to the model.

4. Proposed Methodology

4.1 System Overview

The suggested system is an extension of the three-stage pipeline of the predecessor that includes a fourth component a Secure State Reference Store (SSRS). The general process is based on four steps Prompt Interpretation, Secure State Reference Consultation, Update-Aware Terraform Generation, and Post-Apply State Reconciliation, where the stateless generate-and-apply cycle is replaced with a closed loop that is conscious of, but never directly visible to, the complete sensitive state.

4.2 Secure Minimal State Referencing Model

Each successful apply causes the system to derive and store a de-identified summary of the reduced form of the raw terraform.tfstate file instead of persisting it or sending it to the LLM. This summary will capture: the type of resource (e.g. aws_instance, azurerm_storage_account, or google_compute_instance), the number of resources of that type, the cloud provider, and a locally generated, non-reversible logical name (e.g. web-server-1) that is only used as a reference within a session. There are no IP addresses, ARNs, endpoints, account identifiers, or names based on the actual infrastructure of the user. The translation of a logical alias to the actual address of a Terraform resource is stored at the encrypted Terraform backend, and never in the prompt context delivered to the LLM.

This design is asymmetric on purpose: the LLM is informed about the types of things and their numbers, which is enough to be able to reason about whether a new prompt is describing a change in an existing infrastructure or something completely new, but it is never informed about which particular instance, address, or account that the resource is in. Solution to a logical alias of a concrete Terraform resource address occurs locally, outside the LLM, just prior to the plan/apply stage.

4.3 Prompt-to-Update Terraform Generation

Each time a user makes a prompt, the system retrieves the current minimal state summary and makes it structured context with the natural-language request. The LLM is asked to label the intent as create, modify, query or destroy and to produce Terraform. To a modify intent that corresponds to a known resource type and alias, the system constructs a terraform import block (resolved locally, to the actual resource address) along with the updated attributes only, as opposed to a complete redefinition of the resource. Lifecycle restrictions like prevent destroy and ignore changes are automatically added where the declarative model of Terraform allows it so as to minimize the possibility of accidental recreation. When there is no entry in the state summary corresponding to a create intent, the system reverses the previous generation path of the previous create intent.

4.4 State Reconciliation and Change Detection

Each terraform apply triggers a read of the new terraform.tfstate in the secure backend by the system, which recreates the minimum state summary, recalculating the number of resources and re-compute logical aliases as necessary. This reconciliation step is what allows the LLM to be invoked repeatedly onan up-to-date, but still de-identified, picture of what has actually been deployed, completing the circle between what was actually deployed and what the model thinks is there.

4.5 Security and Privacy Considerations

- The Terraform state file itself is stored only in an encrypted remote backend (e.g., an object store protected with server-side encryption and a managed key), consistent with standard Terraform state-security guidance [12].
- Access to the real resource-to-alias mapping is restricted to the local execution environment and is never included in LLM prompts or logs.
- Least-privilege IAM roles are used for every provider so that the automation component can only act on the resource types it is authorized to manage.
- All logical aliases are generated locally and contain no derivative of real resource identifiers, preventing re-identification through the alias alone.

This design protects the LLM pipeline by minimizing what is sent to it in the first place, rather than by cryptographically hiding a full prompt from an untrusted provider. Confidential prompting takes the latter approach, using a confidential virtual machine and secure partitioned decoding so that even a complete, sensitive prompt never leaves a hardware-protected enclave in readable form [25]. The two approaches are complementary rather than competing: data minimization removes the need to protect information that was never sent, while confidential computing protects whatever is sent regardless of its sensitivity. Section 9 discusses combining both approaches in future iterations of this system.

5. SYSTEM ARCHITECTURE

The architecture comprises five components that are in a closed loop, and there are two trust zones. This is shown in Figure 1. (1) The Prompt Interface receives natural-language input entered by the user; it is linked to the LLM GenerationEngine to create the Generation Plane, which does not have access to sensitive infrastructure information. (2) The Secure State Reference Store stores the de-identified resource-type/count summary in Section 4.2, and is the sole artifact returned to the Generation Plane.(3) The LLM GenerationEngine, trained on the Terraform configuration data as in the prior system [1], is given the prompt and the state summary and generates full or incremental Terraform configuration (candidate HCL). (4) The Terraform ExecutionEngine executes the local alias resolution then executes the standard init, import (when required), planand apply lifecycle within an isolated workspace; with the Encrypted State Backend and the Reconciliation Module, it comprises the Trusted Control Plane. (5) The Reconciliation Module reads the resulting state file of the encrypted backend and updates the Secure State Reference Store and the loop continues. This configuration prevents file data of sensitive state from passing to the Trusted Control Plane, which does not relay to the LLM GenerationEngine.

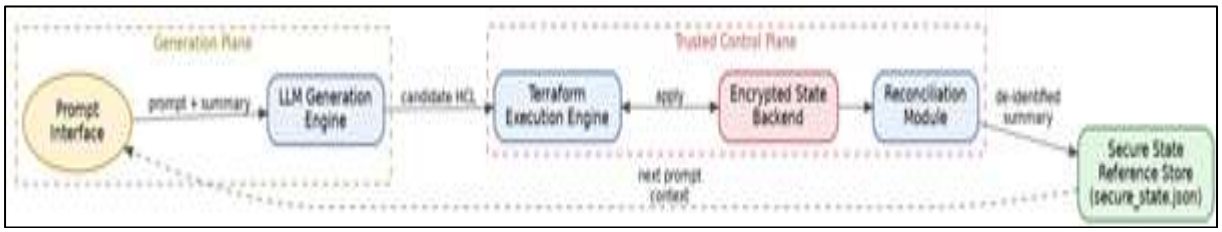

Figure 1: Closed-loop system architecture. The Generation Plane (Prompt Interface, LLM Generation Engine).

never receives sensitive data; only a de-identified summary crosses back from the Trusted Control Plane (Terraform Execution Engine, Encrypted State Backend, Reconciliation Module) via the Secure State Reference Store.

Table 1 summarizes the function of each component and its access to sensitive data, corresponding directly to the two trust zones shown in Figure 1.

Table 1: System Components and Data Access

Component	Function	Access to Sensitive Data
Prompt Interface	Accepts natural-language requests from the user.	No
Secure State Reference Store	Stores the resource type/count summary and logical aliases.	No
LLM Generation Engine	Generates Terraform code from the prompt and the state summary.	No
Terraform Execution Engine	Resolves aliases locally and runs init / import / plan / apply.	Yes (local only)
Reconciliation Module	Reads the real state and updates the de-identified summary.	Yes (local only)

6. IMPLEMENTATION

The installation is based on a toolchain of the previous system. Terraform CLI continues to be the executionengine of all the provisioning activities in AWS, Azure, and GCP. Remote State terraform.tfstate is set up with provider-native encryption at rest (such as an S3 bucket with SSE-KMS, or ananalogous Azure Blob or GCS setup) and access control via provider IAM policies. The implementation of the Secure State Reference Store is as a lightweight structured store (a small JSON or key-value store is enough, as it only stores counts, types and aliases, but not actual infrastructure records) which is re-created after each apply by the Reconciliation Module using the state-inspection commands of Terraform itself. The LLM component is the above-mentioned fine-tuned model, with the extra context field (the state summary) and an intent-

classification instruction, since it is able to differentiate between modification requests and new-resource requests prior to code generation.

This implementation is traced end to end over two sequential prompts in Figure 2. The initial prompt is created and executed with no existing state to reference, then the authoritative terraform.tfstate file produces the secure-state.json (the de-identified alias/provider/type/count summary sent to the LLM) and alias/map.json (the trusted, local-only mapping between alias and real Terraform address, never sent to the LLM). The second response is then responded to with safe context of secure_state.jsonand then state reconciliation is recreated to generate the summary again in the next loop.

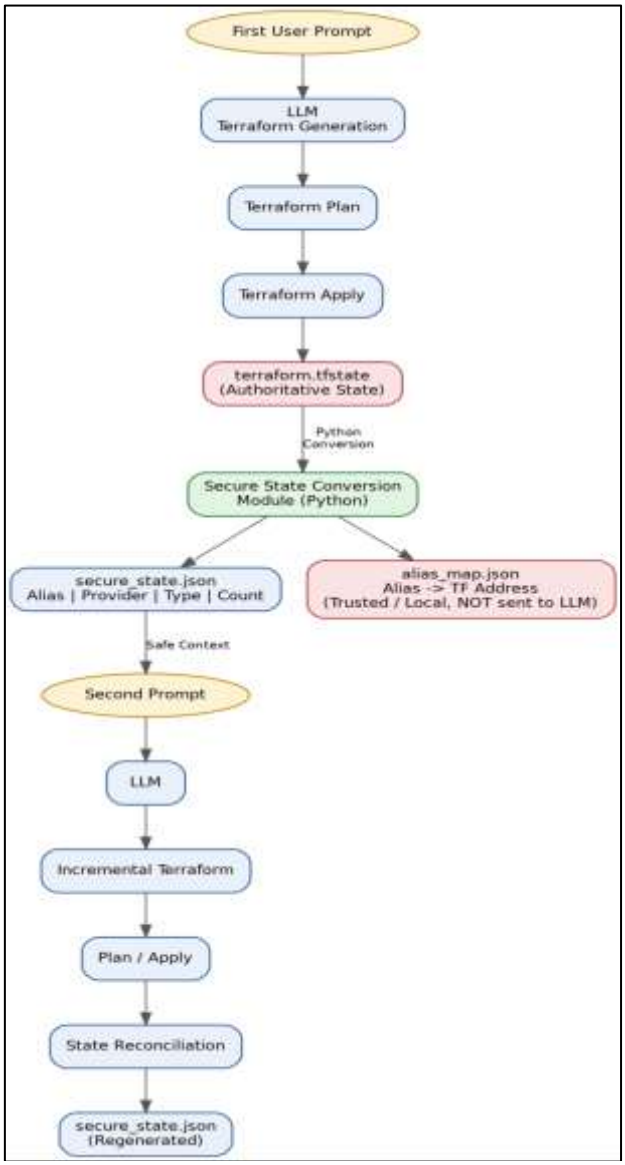

Figure 2: Implementation flow across two consecutive prompts, showing derivation of secure_statejson and alias_mapjson from terraformtfstate by the Secure State Conversion Module, and reuse of the regenerated summary as safe context for the next prompt.

7. Results And Discussion

The proposed design was conceptually tested against the destructive-recreation problem as having been observed as a constraint of the predecessor system, and tested end to end on a live AWS deployment sequence via the prototype console. The situation is similar to that presented in Section 6: a user initially creates an S3 bucket, after which he/she makes two subsequent requests which allow versioning of the same bucket and the creation of a second, separate bucket. Figures 3 to 8 step through this order; Table 2 then sums up the resulting result had the same three prompts been given with respect to the forerunner, stateless system.

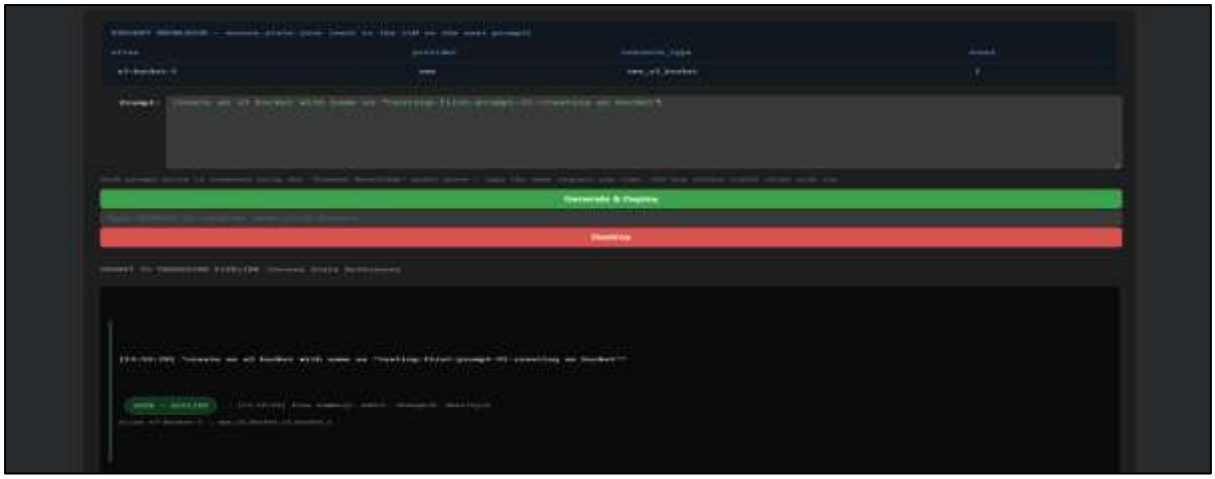

Figure 3: First prompt. The present-knowledge panel is empty before any deployment.

the prompt requests creation of an S3 bucket named "testing-first-prompt-01-creating-an-bucket." The pipeline log confirms a safe apply with add=1, change=0, destroy=0, and records the alias mapping s3-bucket-1 → aws_s3_buckets3_bucket_1.

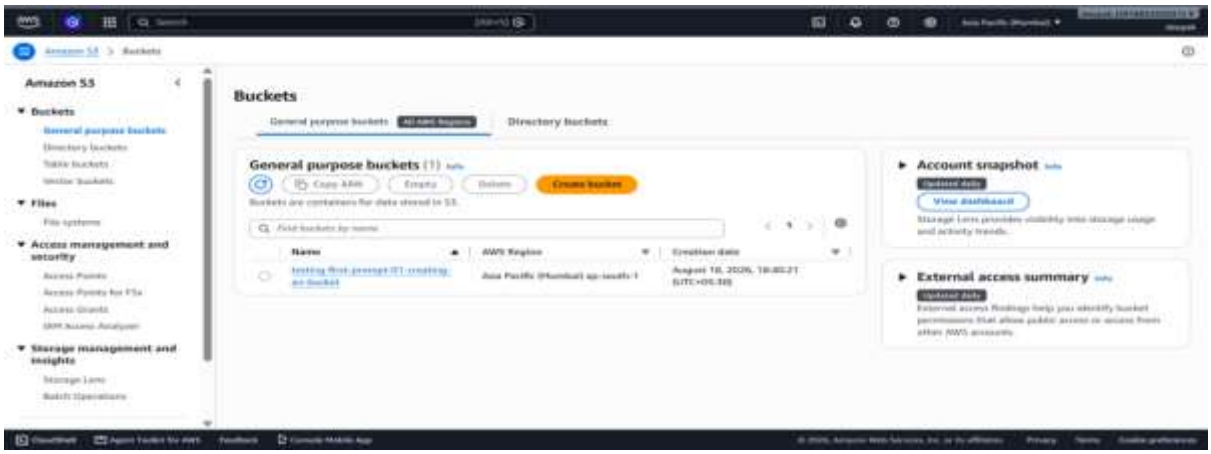

Figure 4: AWS Console confirming the first S3 bucket was created in the ap-south-1 (Asia Pacific – Mumbai) region, matching the name generated from the first prompt.

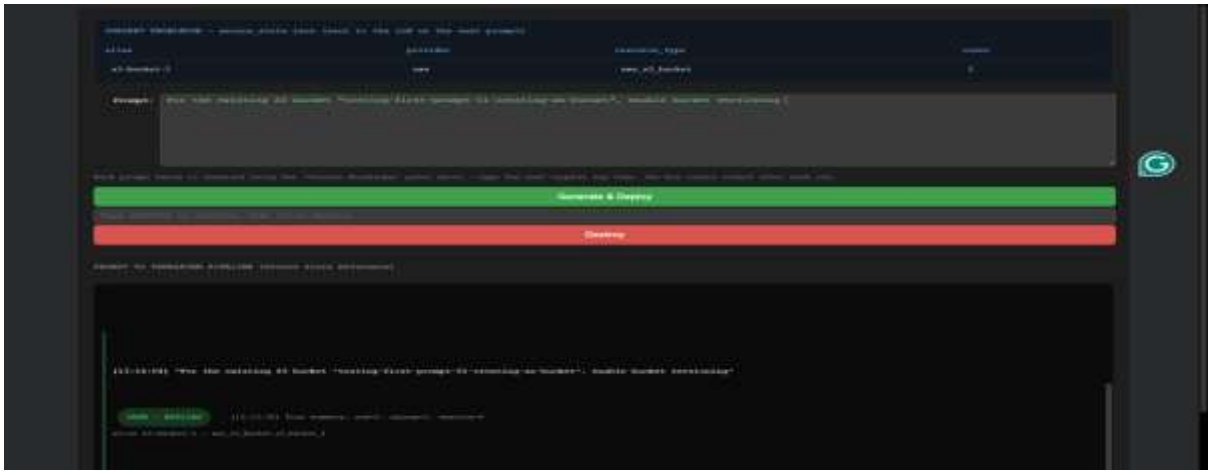

Figure 5: Second prompt. The present-knowledge panel now shows one tracked resource (s3-bucket-1, count = 1).

The prompt asks to enable versioning on the existing bucket by name; the pipeline log reports a safe apply with change=1 and no add or destroy operations, confirming the bucket was modified rather than recreated.

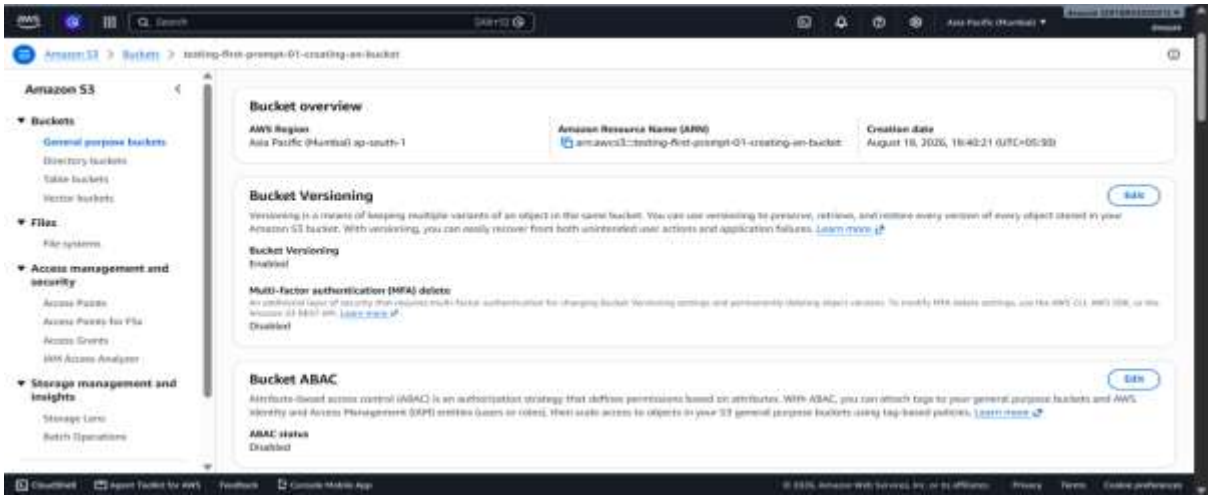

Figure 6: AWS Console confirming that Bucket Versioning was enabled on the existing bucket — the same ARN and creation date persist from Figure 4, verifying the resource was not destroyed and recreated.

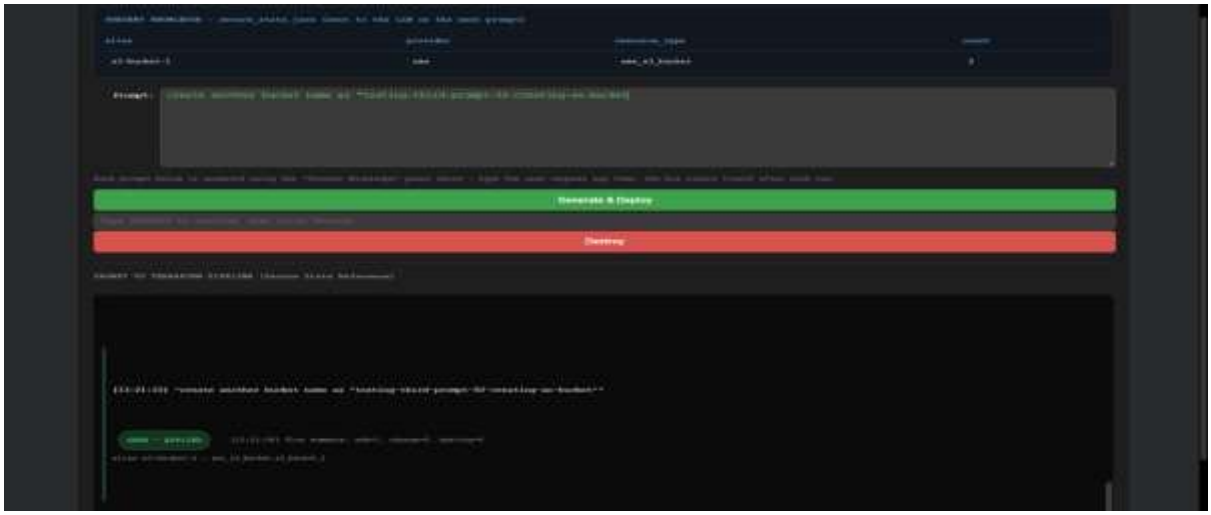

Figure 7: Third prompt. The present-knowledge panel reflects the updated count (s3-bucket-1, count = 2) after the second bucket is provisioned. The pipeline log shows add=1, change=0, destroy=0, and assigns a new alias, s3-bucket-2, to the newly created bucket.

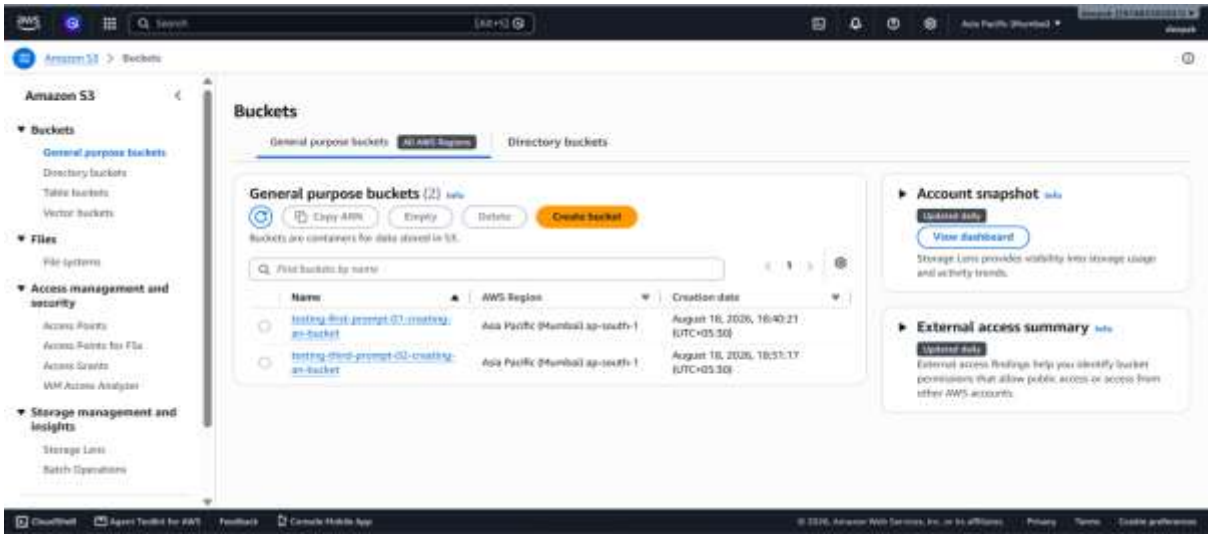

Figure 8: AWS Console showing both S3 buckets present after the third prompt, confirming the first bucket was preserved throughout the second and third interactions rather than being destroyed and recreated.

Table 2: Predecessor System vs. Proposed System Behavior for the Same Prompt Sequence

Step	Predecessor System (No State Reference)	Proposed System (Secure State Reference)
Initial provisioning	S3 bucket "testing-first-prompt-01-creating-an-bucket" created from the first prompt.	S3 bucket created; secure state/summary updated to 1 tracked S3 resource.
Follow-up prompt: enable versioning	New Terraform configuration generated without previous-state awareness; existing bucket configuration may be regenerated rather than updated incrementally.	Intent classified as modify; existing S3 bucket identified from present knowledge and versioning enabled without recreating the bucket.
Third prompt: create another S3 bucket	New Terraform configuration generated from scratch; the previously created bucket may not be preserved if the generated configuration omits it.	Intent classified as create; existing bucket retained and a second S3 bucket, "testing-third-prompt-02-creating-an-bucket," added.
Resulting resource state	Previous infrastructure can be lost or recreated because there is no persistent state reference.	Two tracked S3 resources maintained in the secure state after the third prompt.
Existing resource preservation	The existing bucket is not reliably available to the next prompt.	"testing-first-prompt-01-creating-an-bucket" preserved while the second bucket is created.
Resource recreation risk	High each prompt can result in a newly generated Terraform configuration.	Low previously tracked resources are referenced before generating the next change.
State awareness	No persistent present-knowledge reference.	Secure state/summary provides the LLM with the current resource type/count and tracked-resource knowledge.
Sensitive data sent to the LLM	No state reference; therefore no update-awareness.	None by design only the required resource summary is provided rather than the complete Terraform state.

Table 2 comparisonand the live sequence illustrations in Figures 3 through 8 reveal that the secure minimal state-referencing model is a direct solution to the destructive-recreation problem that does not compromise the original system core security property of not revealing raw infrastructure data to the LLM. Since the LLM never gets any additional information other than resource type, resource count, and a non-identifying logical alias, the amount of information it has available to it to reason about intent is carefully curtailed; this trade-off is explained in more detail in Section 9. In those cases, where only simple, single-resource changes are required - the most frequent case, informally, in the testing of the predecessor system - the type/count/alias summary was adequate to properly distinguish between create and modify intent, as indicated by the change=1 output in Figure 5 and add=1 output in Figure 7.

8. Discussion

The results in Section 7 support several benefits of the proposed design relative to the predecessor, stateless system.

- Idempotent operation Repeated or other similar prompts do not always revisit the entire recreation of resources, as shown by change only apply in Figure 5.
- Less risk of data loss of stateful resources, like storage buckets and databases, as the current resources are imported and updated instead of being destroyed.

- Reduced sensitive-data exposure: the IP addresses, ARNs, endpoints, and account identifiers are never sent to the LLM, resolving the data-leakage issue in the earlier work and reiterated by the current security-mindful benchmarks of LLM-generated IaC [21], [23].
- Incremental updates are done more quickly than full re-provisioning, as only the updated attributes are scheduled and implemented.
- Better auditability, because the Reconciliation Module creates a uniform, de-identified record of resources counts over time.

These advantages are to be read in conjunction with the errors of commission caveats noted in Section 2: since even powerful general-purpose LLMs produce correct Terraform with low rates [22], the safe-apply checks that are visible in Figures 3, 5 and 7 (add/change/destroy counts verified prior to taking action) are a complement, but not a substitute, to the state-referencing model itself. Section 9 discusses the limitations of the current design, and how they could be overcome.

9. FUTURE SCOPE

The present design has several known limitations, which also define the most direct directions for future work.

- The matching of resources by types and numbers is a heuristic; in a world where there are many resources of the same type, alias disambiguation might need further confirmationon the part of the user. Future-work involves the addition of lightweight and nonidentifying relationship metadata (such as security group attached to compute instance) to the Secure State Reference Store, in order to enhance disambiguation of multi-resource changes without reintroducing sensitive identifiers, similar to the dependency-aware repair context in TerraRepair [24].
- It still relies on the native Terraform ability to import, which does not work with all resources types in all three providers.
- Multi-resource changes involving multiple resources that are dependent are more difficult to categorize appropriately based on a type/count summary by themselves than single-resource changes; multi-agent decomposition models like the CloudOps framework by Parthasarathy et al. [17] propose one possible way to treat these cases.
- The design has been made secure, requiring proper configuration of the encrypted remote backend; otherwise, it would re-introduce the risk of data-exposure inherent in the initial design.
- Just like the previous system, LLM hallucination can still happenon the types of resources or attributes that are not well represented in the training data, which is consistent with the low pass rates observed with Terraform specifically in recent benchmarks [22], [23].
- The Reconciliation Module might have an automated drift detection overlay to identify out-of-band modifications outside the prompt-driven pipeline, based on scanner-grounded verification techniques like those in TerraRepair [24].
- Role-based access control might be expanded to allow a multi-tenant usage, and cost-estimation can be offered in the pre-apply plan step to allow users to see an estimation of the spend before approving an update, as proposed in the next version of the predecessor system [1].
- The current data-minimization architecture and hardware-based confidentiality measures like confidential prompting are independent of each other; an improved system could use the already-minimized state summary to run an additional confidential-computing channel as well, such that even the type/count/alias data is concealed in the infrastructure of the LLM provider rather than simply restricted in subject matter [25].

10. Conclusion

This paper has suggested a secure, minimal state-referencing model that addresses the destructive re-provisioning constraint that was found in previous prompt-based multi-cloud automation research. The system uses persisting only de-identified resource type, count, and locally scoped logical aliases, instead of the complete Terraform state file, to provide the LLM enough context to determine the modification requests versus new-resource requests, and produce update-aware Terraform code, without leaving sensitive infrastructure identifiers like IP addresses, ARNs, and endpoints in the prompt pipeline at all. A conceptual test against the self-documented limitations of the previous system and a live three-prompt deployment sequence of the AWS demonstrates that idempotency, risk of data loss, and security posture have been improved, without jeopardizing the initial purpose of the system simple, natural-language-based infrastructure management to both expert and non-expert users.

Funding

The authors declare that no funding was received for this research.

References

[1] Pali Deepak Phani Krishna, & Pachala, S. Prompt based complete Automated multi Cloud Service Management. Department of CSE, Koneru Lakshmaiah Education Foundation.

[2] Ray, B., Chakraborty, S., Ahmad, W. U., & Chang, K. (2021). PLBART: Unified pre-training for program understanding and generation. NAACL-HLT 2021.

[3] Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., & Sutton, C. (2021). Program synthesis with large language models. arXiv preprint arXiv:2108.07732.

[4] Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., et al. (2020). Language models are few-shot learners. Advances in Neural Information Processing Systems (NeurIPS 2020).

[5] Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., et al. (2022). PaLM: Scaling language modeling with pathways. arXiv preprint arXiv:2204.02311.

[6] Dettmers, T., Pagnoni, A., Holtzman, A., & Zettlemoyer, L. (2023). QLoRA: Efficient finetuning of quantized LLMs. arXiv preprint arXiv:2305.14314.

[7] Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., & Zhou, M. (2020). CodeBERT: A pre-trained model for programming and natural languages. Findings of EMNLP 2020.

[8] Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. de O., Kaplan, J., et al. (2021). Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374.

[9] Masola, D. (2023). InfraCopilot: Conversational infrastructure-as-code editor.

[10] Pujar, S., Buratti, L., Guo, X., Dupuis, N., Lewis, B., Suneja, S., Nalawade, A., Jones, M., Morari, A., & Puri, R. (2023). Automated code generation for information technology tasks using Ansible Wisdom.

[11] Zhang, K., Pan, C., Zhang, Y., Xing, Z., & Sun, Y. (2025). Deployability-centric infrastructure-as-code generation: An LLM-based iterative framework.

[12] HashiCorp. (2024). Terraform state documentation: Purpose, sensitive data, and remote backends. HashiCorp Developer Documentation. https://developer.hashicorp.com/terraform/la.nguage/state

[13] Brikman, Y. (2022). Terraform: Up & Running (3rd ed.). O'Reilly Media.

[14] OWASP Foundation. (2023). Data minimization and secrets management guidance. OWASP Cheat Sheet Series.

[15] Bishnoi, P. K., Kumar, D., & Bhanti, P. (2024). Terraform framework development for multi-cloud Docker migration. Journal of Electrical Systems, 20(2s), 1416–1426.

[16] Chanus, T., & Aubertin, M. (2023). LLM and infrastructure as code use case. arXiv preprint arXiv:2309.01456.

[17] Parthasarathy, K., Vaidhyanathan, K., Dhar, R., Krishnamachari, V., Muhammed, B., Kakran, A., et al. (2025). Engineering LLM powered multi-agent framework for autonomous CloudOps. arXiv preprint arXiv:2501.08243.

[18] Mulder, J. (2023). Multi-Cloud Strategy for Cloud Architects. Packt Publishing.

[19] ISO/IEC 22123-1. (2023). Information technology — Cloud computing — Part 1: Vocabulary for multi-cloud architecture and deployment. International Organization for Standardization.

[20] Nijkamp, E., Pang, B., Hayashi, H., Tu, L., Wang, H., Zhou, Y., Savarese, S., & Xiong, C. (2022). CodeGen: An open large language model for code with multi-turn program synthesis. arXiv preprint arXiv:2203.13474.

[21] Srivatsa, K. G., Mukhopadhyay, S., Katrapati, G., & Shrivastava, M. (2024). A survey of using large language models for generating infrastructure as code. arXiv preprint arXiv:2404.00227.

[22] Kon, P. T. J., Liu, J., Qiu, Y., Fan, W., He, T., Lin, L., Zhang, H., Park, O. M., Elengikal, G. S., Kang, Y., Chen, A., Chowdhury, M., Lee, M., & Wang, X. (2024). IaC-Eval: A code generation benchmark for cloud infrastructure-as-code programs. Advances in Neural Information Processing Systems, 37, 134488–134512.

[23] Vargas, F. L. S., Mansilha, R. B., & Kreutz, D. (2026). Security-first evaluation of text-to-Terraform: Benchmarking LLMs and SLMs for secure IaC generation. arXiv preprint arXiv:2608.02672.

[24] Mengistu, M. M., Di Rocco, J., Nguyen, P. T., & Di Ruscio, D. (2026). TerraRepair: A tool-grounded LLM agent for infrastructure-as-code repair. arXiv preprint arXiv:2607.11390.

[25] Gim, I., Li, C., & Zhong, L. (2024). Confidential prompting: Protecting user prompts from cloud LLM providers. arXiv preprint arXiv:2409.19134.