Automated Lifting for Cloud Infrastructure-as-Code Programs

Technical Paper

ABSTRACT

Infrastructure-as-code (IaC) is reshaping how cloud resources are managed. IaC users write high-level programs to define their desired infrastructure, and the underlying IaC platforms automatically deploy the constituent resources into the cloud. While proven powerful at creating greenfield deployments (i.e., new cloud deployments from scratch), existing IaC platforms provide limited support for managing brownfield infrastructure (i.e., transplanting an existing, non-IaC deployment to an IaC platform). This hampers the migration from legacy cloud management approaches to an IaC workflow and hinders wider IaC adoption. Managing brownfield deployments requires techniques to *lift* low-level cloud resource states and translate them into corresponding IaC programs—the reversal of the regular deployment process. Existing tools rely heavily on rule-based reverse engineering, which suffers from the lack of automation, limited resource coverage, and prevalence of errors. In this work, we lay out a vision for Lilac, a new approach that frees IaC lifting from extensive manual engineering. Lilac brings the best of both worlds: leveraging Large Language Models to automate lifting rule extraction, coupled with symbolic methods to control the cloud environment and provide correctness assurance. We envision that Lilac would enable the construction of an automated and provider-agnostic lifting tool with high coverage and accuracy.

1 INTRODUCTION

Cloud computing has become an essential utility across industries [1]. Modern cloud infrastructure offers various types of services—e.g., virtual machines, virtual networks, load balancers, DNS services—which further vary across cloud providers (e.g., AWS vs. Azure vs. GCP). To manage cloud resources, recent management platforms (e.g., Terraform [12]) follow the Infrastructure-as-Code (IaC) design philosophy [18], aiming to replace traditional "ClickOps" and cloud-level API scripting [30]. IaC platforms allow users to define their cloud infrastructure through high-level programming abstractions, masking away complexities of the deployment process [32]. They also aspire to be *cloud-agnostic*—i.e., capable of managing resources in any cloud provider; this is crucial, as many enterprises adopt the multi-cloud strategy to avoid vendor lock in [36]. Terraform is the leading cloud-agnostic IaC platform; at the same time, many other similar solutions exist [2, 4, 9, 10], with different tradeoffs on programming styles and vendor specificity.

While IaC frameworks make it easy to create new infrastructures from scratch, they lack support for porting an existing, non-IaC infrastructure into an IaC program. This need arises in several scenarios. First, many existing infrastructures are constructed using traditional approaches like API scripting, but their DevOps engineers wish to transition them to IaC platforms and modernize their management. Alternatively, some DevOps engineers may prefer using API scripts to construct the infrastructure; but they want to periodically port out an IaC program that describes the underlying resources, which can be scrutinized using compliance checkers for high assurance [6]. Finally, the DevOps team may consist of engineers with different coding preferences, so the overall infrastructure contains components created in IaC and non-IaC approaches. Across these scenarios, the key underlying problem is the same: *Given low-level cloud states from an existing deployment, we need to derive an equivalent, higher-level IaC program that codifies the infrastructure.* This is the reversal of the regular deployment path, for which IaC platforms are not explicitly designed. Hence, this is a challenging process, and in this paper, we call it *IaC lifting*.

```
/* cloud state */
{ // Azure Virtual Machine
 "id":"/<subscription>/…/VM/test",
 "osType":"Linux"
},
{ // Azure Disk (OS disk)
 "id":"/<subscription>/…/disk/OS"
 "manageBy":"…/VM/test"
},{ // Azure Disk (data disk)
 "id":"/<subscription>/…/disk/data"
 "manageBy":"…/VM/test"
}
```

```
/* IaC resource blocks */
resource "azure_linux_VM""test"{
  storage_os_disk {...}
  os_profile_linux_config {...}
  ... ...
}
resource "azure_disk""data"{...}
resource "azure_VM_data_disk_
attachment" "vm-disk" {
  disk_id = azure_disk.data.id
  vm_id = azure_linux_VM.test.id
}
```

Figure 1: Example Cloud State and IaC Program

To understand the complexity of IaC lifting, consider Figure 1 that shows an Azure VM and its attached data and OS disks. In order to lift the cloud-level state (left) into an IaC program (right), the lifting tool must first discover that these resources exist and comprise the totality of the deployment. Next, it needs to reason about resource-level dependencies, recognizing that the VM and disks are attached to each other: while the OS disk should be nested in VM, data disk is managed via "attachment". Finally, it needs to correctly identify all resource attributes, e.g., porting a VM as a Linux or Window resource block in the IaC program. Correctness further requires syntactic checking of the lifted IaC program, as well as checks that ensure its equivalence in terms of resource- and dependency-level state to the existing cloud deployment. This is a delicate procedure, and any slightest mistakes could result in missed resources, undeployable programs, or inequivalent infrastructures.

Industry practitioners have widely recognized the need for IaC lifting, but existing tools [3, 5, 7, 11, 13, 14] struggle with this task. These tools are built upon manually crafted heuristics specialized to a limited set of resource types in specific cloud providers. Lifting is performed using hardcoded rules for discovering resources, reverse-engineering their dependencies, and extracting attribute values from cloud states. Developing such tools requires extensive efforts, since the trial-and-error process involves hand-tuning, manual testing, and often, some amount of guesswork. Maintaining these tools incurs additional burden, since supporting new resources or providers requires further manual engineering. As a result, today's tools are brittle and do not offer any correctness assurance, which means the lifted programs might contain a variety of errors. While devoting more engineering hours could lead to improvements, scaling the current practice to multiple cloud providers is inherently challenging. This chips away at the ideal of IaC-style management.

Lilac aims to remove the extensive, error-prone manual engineering in current IaC lifting practices. Our vision is to automatically perform cloud-agnostic IaC lifting with both high coverage and accuracy, with the help of LLM agents. The key insight is that, *IaC lifting rules (the reverse direction) can be learned from observing IaC program deployments (the forward direction).* Since IaC

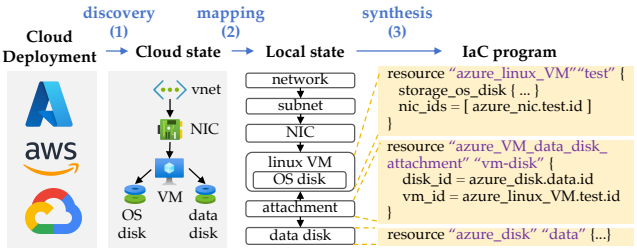

Figure 2: Typical IaC Lifting Workflow

frameworks are well-engineered for translating IaC programs to cloud-level resource states, AI-powered intelligence could closely observe this workflow and infer the backward mapping from cloud states to IaC programs. We envision a self-evolving agent that takes available IaC programs as inputs, deploys them into the cloud to obtain resource states, and then rolls out a sequence of experiments to find, port, and translate resource states back to the original IaC programs. This approach offers the following benefits: (1) if a resource type is witnessed in the input IaC corpus, the agent will learn how to lift it; (2) lifting rules can be verified by comparing lifted IaC programs with the original input programs; (3) lifting rules for each resource type can be progressively improved as we acquire more input programs for training the agent. Above all, this is an automated process, and one that can be applied across cloud providers, without repeated development burden for each cloud.

Lilac uses a neurosymbolic approach to IaC lifting, combining the power of LLM agents for knowledge retrieval (e.g., from online cloud documentation) and symbolic methods for reliability guardrails (e.g., for distilling lifting rules). Although LLMs have remarkable performance, IaC lifting requires high precision and reliability. LLM can hallucinate, especially in complex tasks that require domain-specific knowledge and interaction with external tooling. Their inference results may not be consistent across sessions, with low reproducibility. Cloud operations, on the other hand, are safety-critical and cannot tolerate imprecision or errors. We design an LLM agent capable of dividing lifting rule extraction into simpler subtasks, each verifiable via complementary symbolic mechanisms. This combination will help Lilac achieve automation with assurance of the quality of generated results.

Lilac is still an ongoing work; but as initial evidence on the effectiveness of our proposed approach, we have built a research prototype capable of extracting several types of important IaC lifting rules, performing on par with or exceeding today's hand-crafted tools. First, we show that LLM agents can be used for a variety of rule extraction tasks, including cloud resource discovery and resource mapping generation. Moreover, we explore the usage of deployment-based testing (e.g., comparing lifted programs against initial programs) for validating the learned rules. Last but not least, we show the learned rules can be used to lift real-world cloud infrastructure. We also propose a roadmap towards an end-to-end, industry-strength pipeline for IaC lifting across cloud providers, which could deal with more fined-grained rules and ensure correctness of lifted programs for Azure, GCP, AWS, and other clouds.

2 MOTIVATION

In this section, we further motivate our problem by delving into its use cases, requirements, and solution space.

Tools	Azure	Google	AWS	Coverage
TerraCognita [11]	✔	✔	✔	~10%
Terraformer [14]	✔	✔	✔	~10%
aztfexport [5]	✔	✖	✖	~95%
gcloud export [7]	✖	✔	✖	~30%
aws2tf [3]	✖	✖	✔	~60%

Table 1: Scope and Coverage[1] of Existing IaC Lifting Tools

2.1 Lifting: The Road Less Traveled

Infrastructure-as-Code (IaC) tools such as Terraform [12] simplify cloud management tasks by providing an abstraction layer that hides away cloud operation details. Terraform compiles a user-specified IaC program and compares it with "local states," which are initially empty, to generate a provider-agnostic deployment plan. It then invokes provider-specific plugins based on the plan, issuing a sequence of API calls to deploy the constituent cloud resources. As the last step, Terraform updates the local states with the deployed cloud states to be in sync. This "forward" workflow, from user programs to cloud states, is what IaC tools are built for.

The backward path for lifting "in the wild" cloud states—i.e., an infrastructure that is not constructed using IaC tools—back to local states and IaC programs, is less studied. Figure 2 depicts the lifting workflow: (1) obtaining cloud states from the provider backend, (2) mapping cloud states back to local states, and (3) translating local states into IaC programs. Our survey with IaC users and developers indicates that a lifting workflow should offer the following features, none of which are fully covered by existing tools:

(1) *Resource coverage*: the IaC lifting process should be able to handle all constituent cloud resources, across cloud providers, without any missed resources or dependencies.
(2) *Correctness*: the lifted program should compile correctly; when deployed to the cloud, it should produce the same cloud states as those of the brownfield deployment. A correct program should be able to pass through multiple IaC-native verifications (§3.2.2).

2.2 Existing IaC Lifting Tools

Historically, IaC platform developers have attempted to add official support for lifting, with Terraform import [13] being the most notable efforts. Essentially, Terraform import partially automates step (2) and (3) in Figure 2 by translating local states to IaC resource blocks. However, to maintain the provider agnostic interface, it asks users to obtain cloud states from providers, identify the type mapping between cloud states and local states, then fix a wide range of errors caused by the plain porting from local states to IaC resource blocks. Since IaC platforms struggle to offer automated lifting support, many third-party tools have been developed on top of them to improve the status quo. As highlighted in Table 1, existing lifting tools fall into two categories.

(1) *Cloud-agnostic* lifting tools like TerraCognita support multiple major cloud providers, but suffer from low resource coverage within each cloud, which means they will simply ignore all resources except the most popular ones.

[1]Coverage is calculated as # Supported Terraform resource types by the tools / # Total Terraform resource types in the cloud providers.

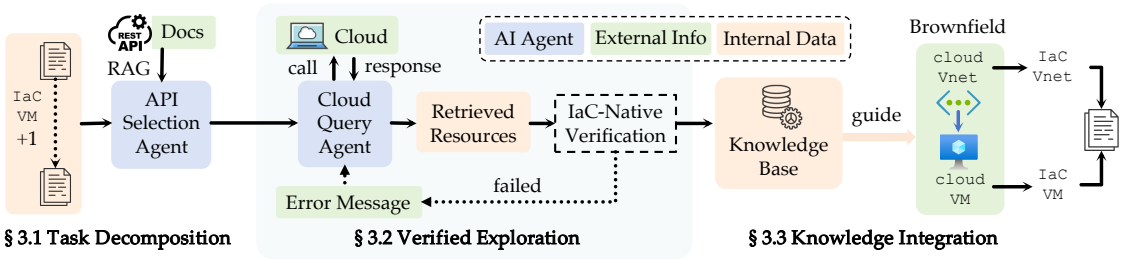

Figure 3: LILAC Workflow

(2) *Cloud-specific* lifting tools like Azure's aztfexport achieve higher coverage as they have tighter coupling with cloud providers. Nevertheless, such lifting tools are inherently limited in their scopes, as their implementation cannot easily generalize to other providers.

2.3 Key Idea and Challenges

Throughout this paper, we discuss how to learn IaC lifting rules by observing the IaC deployment process. Our key idea is that, given both the original IaC programs and their corresponding cloud states, the lifting process is equivalent to translating cloud states back to exactly the same original programs. However, finding the appropriate translation is a complex exploratory task that cannot be managed by manual efforts or simple heuristics. Instead, we leverage the capabilities of emerging AI agents, while addressing domain-specific challenges that arise.

(1) *Complexity of cloud environments.* Lifting rule extraction faces a combinatorial search space. On one hand, IaC programs contain many inter-connected, hierarchical IaC resources, each with their own attributes. On the other hand, cloud providers offer thousands of RESTful APIs, each of which contains information useful to some part of the lifting task. A naive application of LLM would struggle to build connections between the two, resulting in low coverage.

(2) *Inaccuracy caused by hallucination.* Lifting rule extraction is a safety-critical task. Incorrect rules introduced by LLM hallucination could directly lead to incorrect lifting results, which might take a large amount of manual efforts to fix. If users fail to notice or fix some of these errors, then the lifting results could contain *resource drifts* where IaC resources do not match with cloud states, jeopardizing the reliability of the entire cloud infrastructure.

(3) *Ambiguity within learned results.* Lifting rule extraction has stringent requirements on the format of final and intermediate results. Given that our goal is to directly plug learned rules into a symbolic lifting engine, all the results generated by LLM must follow the grammar of lifting engine, without any inconsistencies or ambiguity. Failing to do so would necessitate manual processing of the learned rules, which negates the benefit offered by our automated pipeline.

3 SOLUTION SKETCH

In this section, we outline the roadmap for LILAC, a fully automated IaC lifting rule extraction pipeline. Figure 3 illustrates the workflow of our three-phase system design. The pipeline begins with a set of IaC programs that initiate greenfield deployment tests. In the *Task Decomposition Phase*, LILAC breaks these programs into incremental tests to control cloud environment updates and guides API selection through an agent. During the *Verified Exploration Phase*, the Cloud Query Agent interacts with the cloud environment via APIs, retrieves resources in IaC format, and verifies their correctness using IaC-Native Verification. Finally, verified observations are condensed into generalizable rules stored in the knowledge base, enabling future lifting of brownfield deployments to IaC.

3.1 Task Decomposition Phase

Given input IaC programs and candidate cloud APIs, LILAC firstly decomposes the task of learning the relation between IaC programs and available API calls by (1) divide-and-conquer resource blocks within each IaC program, and (2) filtering out API calls that are irrelevant to the target resources.

3.1.1 Incremental Resource Deployment. The mapping between IaC resource blocks and cloud states is highly asymmetric and case-by-case. There is no one-for-all mapping between the modification in IaC program and the resulting cloud state update. As an example, while adding a virtual machine in Terraform simply corresponds to provisioning a new virtual machine in Azure, creating a VM-data disk attachment block does not create any new Azure resource instances. Instead, it updates the `storageProfile` property of the VM with information on the newly attached disk. Consequently, feeding entire IaC programs directly into the pipeline makes it hard to distinguish the impact of each IaC resource blocks on cloud states, which hampers the reverse path learning process. LILAC mitigate this problem by deploying IaC resources within each program incrementally, allowing our pipeline to find the exact mapping between IaC resource deployment and cloud state updates. To achieve this, we transform each input Terraform program into a sequence of transition steps, with each step adding a single resource to the cloud. Essentially, we perform topological sorting among resources based on their dependencies, to decide which resources should be created before others during incremental deployment.

3.1.2 API Selection Agent. Given an incrementally deployed IaC resource block, our cloud API selection agent predicts cloud API groups that are relevant to the target resource. This is a necessary step so that the pipeline does not get lost on among thousands of available APIs. To achieve this, we construct the agent using Retrieval-Augmented Generation (RAG), which matches target resource against the text descriptions and usage examples of cloud

(A) Cloud Type-API Map

Cloud Type	Avaliable APIs
Microsoft.Compute/VirtualMachines	`list vms`

(B) API Tree Map

Parent API	Child API	Arguments
`list vms`	`list disks under a vm`	VM.name

(C) Response Inference Map

Infer IaC Type	Schema
VM	VM.id from `list vms`
VM-datadisk attachment	VM.id from `list vms` plus disk.name from `list disks under a vm`

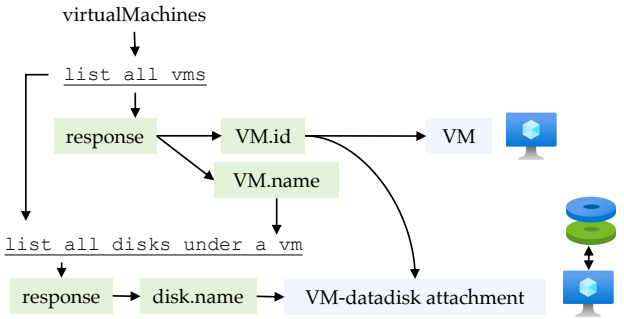

Figure 4: Example Scenario of Brownfield Deployment Lifting: VM, datadisk and VM-datadisk attachment in Azure

API documentation to find the most relevant API groups. It further ensures the number of APIs in each group stays within the LLM's context windows for more accurate model output. This step is crucial because there is no clear, predefined mapping between IaC types and cloud service categories or API families. Current solutions require developers to manually parse extensive API documentation to select suitable options, which is highly inefficient.

3.2 Verifiable Exploration Phase

Given a deployed IaC resource and its relevant cloud API groups, Lilac initiates an exploration phase to experiment on how cloud APIs and their responses could help with lifting cloud states. We also incorporate verification to guide the agent for accurate discovery and pave the path for future general rule extraction.

3.2.1 Cloud Query Agent. Given an IaC resource and relevant API group as inputs, an LLM-powered agent explores the cloud environment to retrieve information useful for lifting. Within each iteration, the agent can take one among three actions: 1) calling cloud API sequences to query relevant cloud resources and their child resources; 2) generating the target IaC resource block if the information from API responses is sufficient; 3) requesting the API Selection Agent to select a new API family if no suitable APIs are available. Across these iterations, we record all the actions the agent takes along its exploration path, and feed them as input to the next step verification. As feedback, later verification would return any error messages detailing the location and root cause of errors in the agent's lifted program, with which our Cloud Query Agent continuously refines and updates the query and lifting path.

3.2.2 IaC-Native Verification. Verification is critical to ensuring the correctness of the agent workflow and the extracted lifting rules. Multiple rounds of testing with IaC-native verifiers should be performed to meet various correctness criteria.

(1) *Existence Check*: When a target resource is identified, Lilac attempts to import the corresponding resource into Terraform using the retrieved ID, ensuring that the cloud query agent is not hallucinating nonexistent resources.
(2) *Equivalence Check*: Lilac asks Terraform's backend to synchronize lifted IaC programs with the cloud provider to detect any state drift, ensuring the lifted program correctly describes the current cloud states.
(3) *Redeployment Check*: To check the usability and portability of lifted programs, Lilac performs deployment based testing to observe whether the lifted programs can restore current cloud states from empty.

3.3 Knowledge Integration Phase

In this phase, the agent's unstructured observations from specific resource queries are consolidated into generalizable rules within our dynamic knowledge base. This well-structured rule set supports reproducible lifting from brownfield deployments.

3.3.1 Query Rule Knowledge Base. It consolidates the cloud query agent's observations into structured rules, enabling accurate and reproducible IaC lifting. Initially, the agent records all intermediate steps involved in the successful retrieval of each IaC resource, updating these observations in a dynamic knowledge base. These observations are then transformed symbolically into a formalized set of query rules, comprising the following categories:

(1) *Cloud Discovery Rules* define how to retrieve cloud resources and identify IaC components that can be lifted into the output program. Key elements include: (1) *API Query Chain*: sequences of APIs that navigate the hierarchical cloud topology to locate target resources. (2) *Cloud-IaC Resource Inference*: A mapping of specific cloud resources and their properties to corresponding IaC resource types, enabling accurate identification of liftable components.
(2) *Cloud Mapping Rules* handle the detailed mapping of observed cloud resources to their IaC representations. For instance, *Attribute-level Lifting Condition* determines which components should be represented in the IaC output. As illustrated in Figure 2, when an OS disk and a data disk are discovered, only the data disk is generated as a standalone IaC resource, while the OS disk is nested inside the VM. This avoids redundancy and ensures synchronization with the cloud state.
(3) *Dependency Restoration Rules* focus on re-establishing the connections between the IaC resource and form hierarchical structure of IaC topology. Unlike existing tools that hardcode resource IDs (e.g., aws2tf), which prevents redeployment of infrastructure since old IDs are invalidated once destroyed, Lilac leverages chained cloud query API calls and analyze nested attributes in API responses. Mined rules accurately represents hierarchical topologies and ensures the lifted program can manage dependencies and maintain portability across deployments.

Tools	aztfexport	Lilac
False Positive (redundancy)	4.6%	2.3%
False Negative (oversight)	7.0%	0

Table 2: Lifting Error Rate of aztfexport and Lilac

3.3.2 Brownfield Deployment Lifting. When lifting a brownfield deployment, Lilac relies solely on the query rules stored in its knowledge base. This means that once an IaC resource has passed through the previous two phases, we can lift the same type when encountering it in future lifting.

An example scenario in Figure 4 illustrates how to leverage those rules. In the resource group to be lifted, there is a VM and its associated data disk. To represent this topology in Terraform, we need resource blocks for both the VM and datadisk, along with a separate attachment declaration. The process begins by querying the top-level resources in the group and identifying the relevant cloud APIs corresponding to the cloud resource types, as defined in the "Cloud Type-API Map". For instance, we might pick out the `list vms` to query details of all VMs in the group. Once we obtain the API response, typically a JSON object, we parse it using the inference rules from "Response Inference Map". For example, the "VM.id" entry indicates a VM in the IaC context, as well as the parent resource ID needed for a potential data disk attachment. Next, using the "API Tree Map", we move to the relevant child API, `list disks under a vm`, with the required argument VM.name extracted from the previous response. From this subsequent API response, we inferred the full ID of the VM-datadisk attachment in the IaC configuration.

4 PRELIMINARY VALIDATION

In this section, we show that our current prototype could extract many important rules and use them to lift real infrastructure, with results surpassing manually implemented lifting tools. For all evaluation items, we first deploy resources via Terraform, and then use the original programs as lifting ground truth.

RQ 1: Can Lilac outperform existing cloud-agnostic tools? Existing cloud-agnostic lifting tools suffer from narrow coverage, typically supporting only about 10% of resources for each major cloud provider (Table 1). These tools often isolate resources without establishing the necessary connections between them. In contrast, Lilac accurately restores these dependencies. For example, in a deployment where a network is safeguarded by a security group, TerraCognita generates separate IaC descriptions for the two resources, completely ignoring their relationship. By comparison, Lilac includes an association block to accurately represent their connection. Similarly, Lilac support resources such as the association between subnet and route table, as well as attachment between VM and data disk, all of which are overlooked by existing cloud-agnostic tools. Our following comparisons with cloud-specific tools further highlight our advantages over cloud-agnostic tools, as the former already outperform the latter within their respective cloud.

RQ2: Can Lilac achieve results on par with cloud-specific tools? We evaluated 43 popular Azure resources in Terraform, focusing on topology reconstruction, and recorded the error rate for lifted instance numbers in Table 2. Both tools accurately lifted most resources but occasionally produced redundant resource blocks (false positives). For Lilac, these errors stem from incomplete rule mining due to limited input programs, which could be resolved with a more comprehensive dataset. Notably, Lilac successfully lifted all baseline resources, while aztfexport missed some components (false negatives), such as the firewall rule collection for NAT. These results demonstrate that Lilac delivers performance comparable to advanced, specialized tools like aztfexport, with significantly less manual effort.

Category	Terraform(IaC) Type
ComputeNetwork FirewallPolicy	network firewall policy network firewall policy association network firewall policy rule
ComputeRouter	router interface, router peer router NAT, router NAT address
NetworkPeering	network peering network peering route

Table 3: Example List of Lilac Supported Resource out of `gcloud export` Coverage

RQ3: Can Lilac mine the lifting rules that cannot be covered by cloud-specific tools? We tested Lilac on popular resource types that are not supported by the official Google lifting command, the best support for Google cloud so far. Our experiments demonstrate that Lilac successfully lifts these resources, which would otherwise be ignored when using `gcloud export`. Table 3 highlights a selection of resources that Lilac supports but are not handled by Google's tool. In addition, Lilac not only includes these components but also captures the necessary associations, configuring the firewall within the network and specifying detailed rules to manage firewall behavior. In this way, Lilac delivers a more accurate and high-fidelity representation that ensures semantic equivalence with the original infrastructure configuration.

RQ4: How efficiently does the symbolic design assist AI agents in knowledge extraction? In our neurosymbolic solution, we use symbolic methods to guide the exploration process of our Cloud Query Agent. This includes incremental resource deployment to control updates to the cloud environment, and IaC-native verification to validate agent's observation. To evaluate the impact of this design, we conducted an ablation study where we removed these components, transforming the pipeline into a pure LLM-driven approach.

Figure 5 illustrates the percentage of successfully lifted resources in Azure under various configurations. The results indicate that without symbolic guidance, lifting accuracy drops significantly as the size of the input IaC program increases. Removing incremental resource deployment reduces accuracy for larger resource groups due to the increased complexity of managing resource dependencies without structured guidance. Omitting correctness verification substantially impacts accuracy, as agents querying irrelevant resources can pass through the pipeline, resulting in problematic rules for future use. These findings demonstrate that a purely LLM approach is insufficient to support IaC lifting, underscoring the importance of Lilac design for reliable and effective performance.

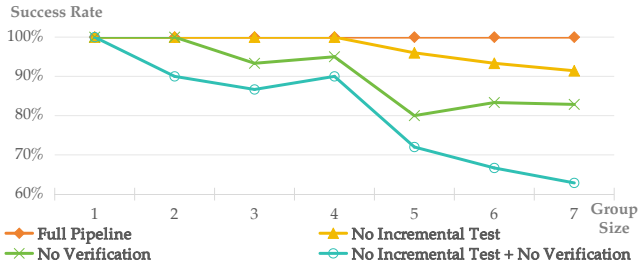

Figure 5: Ablation study of Lilac pipeline

5 RELATED WORK

Cloud Infrastructure-as-Code. IaC has emerged as a leading paradigm for cloud operation management [18, 32]. Recent research on IaC has focused on areas such as security policy enforcement [21, 27] and program testing [33, 34]. Two relevant lines of work include LLM-aided IaC synthesis [26] and semantic rule mining for IaC programs [31]. Lilac extends IaC research by addressing the critical challenge of program lifting, essential for intelligent cloud provision and migration.

LLM-aided cloud management and AIOps. LLM agents are ideal tools for automating cloud operations [23], with recent work exploring IaC code generation [26], cloud incident analysis [35], infrastructure safeguarding [20], cloud API interactions [29], as well as logs analysis [25]. While promising, none of them have touched the demand for IaC lifting.

Program lifting and bidirectional lens. *Verified Lifting* [8], synthesizing high-level programs from low-level counterparts with verified correctness, has been explored in various contexts, such as lifting Java to parallel data processing frameworks [15–17]. Research on *bijective programming lenses* [19, 28] or *bidirectional transformation* [22, 24] further studies programming language constructs for mappings across data formats. Lilac performs a complementary study in the context of cloud management practices.

6 CONCLUSION

Infrastructure-as-Code (IaC) is a leading paradigm in cloud resource management. Emerging IaC platforms allow users to describe and deploy their intended infrastructure with ease. *IaC lifting* is a task that aims at transplanting brownfield non-IaC infrastructure back into IaC programs—a reversal of the traditional IaC deployment workflow. However, existing attempts at IaC lifting struggle to meet diverse user expectations and consume extensive engineering efforts. Our proposed system, Lilac, outlines a vision for *fully automated* IaC lifting. It applies a neurosymbolic approach to automate the lifting rule discovery and utilization with cloud-agnostic design. A full realization of our vision will extend the benefits of IaC to all cloud practitioners, greatly simplifying their resource management, migration and scaling workflow.

REFERENCES

[1] 37 cloud computing statistics, facts & trends for 2024. https://www.cloudwards.net/cloud-computing-statistics/.
[2] AWS CloudFormation. https://aws.amazon.com/cloudformation/.
[3] aws2tf. https://github.com/aws-samples/aws2tf.
[4] Azure Bicep. https://learn.microsoft.com/azure/azure-resource-manager/bicep/.
[5] Azure Export for Terraform. https://github.com/Azure/aztfexport.
[6] Checkov: ship code that's secure by default. https://bridgecrew.io/checkov/.
[7] Export your google cloud resources to terraform format. https://cloud.google.com/docs/terraform/resource-management/export.
[8] Metalift: A program synthesis framework for verified lifting applications. https://metalift.pages.dev/.
[9] OpenTofu: The open source infrastructure as code tool. https://opentofu.org/.
[10] Pulumi: Iac in any programming language. https://www.pulumi.com/.
[11] TerraCognita by Cycloid. https://github.com/cycloidio/terracognita.
[12] Terraform by Hashicorp. https://www.terraform.io/.
[13] Terraform import. https://developer.hashicorp.com/terraform/language/import/generating-configuration.
[14] Terraformer: CLI tool to generate terraform files from existing infrastructure. https://github.com/GoogleCloudPlatform/terraformer.
[15] M. B. S. Ahmad and A. Cheung. Leveraging parallel data processing frameworks with verified lifting. *Electronic Proceedings in Theoretical Computer Science*, 2016.
[16] M. B. S. Ahmad and A. Cheung. Optimizing data-intensive applications automatically by leveraging parallel data processing frameworks. In *Proceedings of the 2017 ACM International Conference on Management of Data*, 2017.
[17] M. B. S. Ahmad and A. Cheung. Automatically leveraging mapreduce frameworks for data-intensive applications. In *Proceedings of the 2018 International Conference on Management of Data*, 2018.
[18] M. Artac, T. Borovssak, E. Di Nitto, M. Guerriero, and D. A. Tamburri. Devops: introducing infrastructure-as-code. In *Proceedings of the 2017 IEEE/ACM International Conference on Software Engineering Companion*, 2017.
[19] D. M. Barbosa, J. Cretin, N. Foster, M. Greenberg, and B. C. Pierce. Matching lenses: alignment and view update. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, 2010.
[20] D. Cao and W. Jun. Llm-cloudsec: Large language model empowered automatic and deep vulnerability analysis for intelligent clouds. In *IEEE INFOCOM 2024 - IEEE Conference on Computer Communications Workshops*, 2024.
[21] C. Cauli, M. Li, N. Piterman, and O. Tkachuk. Pre-deployment security assessment for cloud services through semantic reasoning. In *Proceedings of International Conference on Computer Aided Verification*, 2021.
[22] K. Czarnecki, J. N. Foster, Z. Hu, R. Lämmel, A. Schürr, and J. F. Terwilliger. Bidirectional transformations: A cross-discipline perspective. In *Proceedings of International Conference on Theory and Practice of Model Transformations*, 2009.
[23] B. A. A. for Autonomous Clouds: Challenges and D. Principles. Manish shetty and yinfang chen and gagan somashekar and minghua ma and yogesh simmhan and xuchao zhang and jonathan mace and dax vandevoorde and pedro las-casas and shachee mishra gupta and suman nath and chetan bansal and saravan rajmohan. In *Proceedings of the 2024 ACM Symposium on Cloud Computing*, 2024.
[24] S. Hidaka, Z. Hu, K. Inaba, H. Kato, K. Matsuda, and K. Nakano. Bidirectionalizing graph transformations. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, 2010.
[25] Z. Jiang, J. Liu, Z. Chen, Y. Li, J. Huang, Y. Huo, P. He, J. Gu, and M. R. Lyu. Lilac: Log parsing using llms with adaptive parsing cache. In *Proceedings of ACM International Conference on the Foundations of Software Engineering*, 2024.
[26] P. T. J. Kon, J. Liu, Y. Qiu, W. Fan, T. He, L. Lin, H. Zhang, O. M. Park, G. S. Elengikal, et al. Iac-eval: A code generation benchmark for cloud infrastructure-as-code programs. *Advances in Neural Information Processing Systems*, 2024.
[27] J. Lepiller, R. Piskac, et al. Analyzing infrastructure as code to prevent intra-update sniping vulnerabilities. In *Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2021.
[28] A. Miltner, K. Fisher, B. C. Pierce, D. Walker, and S. Zdancewic. Synthesizing bijective lenses. In *Proceedings of the ACM on Programming Languages*, 2017.
[29] S. G. Patil, T. Zhang, X. Wang, and J. E. Gonzalez. Gorilla: Large language model connected with massive apis, 2023.
[30] F. Petrillo, P. Merle, N. Moha, and Y.-G. Guéhéneuc. Are REST APIs for cloud computing well-designed? an exploratory study. In *Proceedings of International Conference on Service-Oriented Computing*, 2016.
[31] Y. Qiu, P. T. J. Kon, R. Beckett, and A. Chen. Unearthing semantic checks for cloud infrastructure-as-code programs. In *Proceedings of the 29th Symposium on Operating Systems Principles*, 2024.
[32] Y. Qiu, P. T. J. Kon, J. Xing, Y. Huang, H. Liu, X. Wang, P. Huang, M. Chowdhury, and A. Chen. Simplifying cloud management with cloudless computing. In *Proceedings of ACM Workshop on Hot Topics in Networks*, 2023.
[33] D. Sokolowski, D. Spielmann, and G. Salvaneschi. Automated infrastructure as code program testing. *IEEE Transactions on Software Engineering*, 2024.
[34] D. Sokolowski, D. Spielmann, and G. Salvaneschi. Unleashing the giants: Enabling advanced testing for infrastructure as code. In *Proceedings of IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, 2024.
[35] 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.
[36] Z. Yang, Z. Wu, M. Luo, W.-L. Chiang, R. Bhardwaj, W. Kwon, S. Zhuang, F. S. Luan, G. Mittal, S. Shenker, and I. Stoica. SkyPilot: An intercloud broker for sky computing. In *20th USENIX Symposium on Networked Systems Design and Implementation*, 2023.