CSE 599 Report: IaC Lifting and Cloud State Identification

ChunYuan Hsu
University of Michigan
Ann Arbor, MI, USA
chyhsu@umich.edu

I. Lifting

Our work focuses on *Infrastructure-as-Code (IaC) lifting*, a process that converts existing cloud deployments (brownfield infrastructure) into declarative IaC programs such as Terraform. While IaC tools are well designed for forward deployment, lifting is inherently challenging due to the semantic gap between low-level cloud states and high-level IaC abstractions.

In this project, we study practical IaC lifting by mapping cloud-provider-specific JSON representations to Terraform resource blocks, attributes, and dependencies, with a primary emphasis on Azure infrastructure. We systematically analyze lifting challenges including non-1:1 resource mappings, nested resources, missing or redundant JSON fields, implicit dependencies, and attribute normalization.

Using extensive real-world examples—such as virtual machines, DNS, CDN, Kubernetes, and AI services—we categorize lifting scenarios into distinct structural cases. Both cloud states and Terraform configurations are modeled as graphs, where nodes represent resources and edges represent dependencies or references. Based on these graph representations, we derive mapping rules that translate cloud nodes and edges into Terraform resource blocks and inter-resource references.

Our lifting workflow incrementally improves output quality through multiple stages, including brute-force generation, documentation-guided refinement, forward learning of mapping rules from deployments, and program-level refactoring. Validation is performed using Terraform-native commands such as `init`, `validate`, and `plan`. Overall, this work emphasizes engineering feasibility, detailed case analysis, and rule extraction grounded in observed cloud behavior.

II. Lilac

Lilac is an existing research project that also targets the IaC lifting problem, aiming to build an automated, cloud-agnostic, and correctness-aware lifting system. Unlike ad-hoc or rule-based tools, Lilac seeks to generalize lifting across cloud providers by learning reusable lifting rules from forward IaC deployments.

The core idea behind Lilac is to observe how IaC programs are translated into cloud states during deployment, and then infer the reverse mapping. Lilac incrementally deploys IaC programs, queries cloud provider APIs, and compares the resulting cloud states with the original IaC configurations. From these observations, Lilac extracts lifting rules that map cloud resources, attributes, and dependencies back to IaC constructs.

Lilac adopts a neurosymbolic design. Large Language Models (LLMs) are used for exploration tasks such as API selection, documentation retrieval, and pattern discovery, while symbolic techniques and IaC-native verification are employed to enforce correctness, reproducibility, and safety. Verification steps include equivalence checks, redeployment checks, and Terraform-native validation to ensure that lifted programs faithfully represent the original cloud infrastructure.

III. Between Lifting and Lilac

Although our work and Lilac address the same fundamental problem, they differ in scope, abstraction level, and system integration. Our work represents a focused, implementation-oriented lifting process, while Lilac is a complete end-to-end framework designed for automation and generalization.

In terms of scope, our method concentrates on rule learning and graph-based mapping for concrete lifting scenarios, primarily using Azure as the main experimental platform. Lilac generalizes this approach across multiple cloud providers and integrates additional components such as API selection agents, persistent knowledge bases, and cross-run rule reuse.

In terms of abstraction, our method operates at a lower level, explicitly reasoning about JSON schemas, Terraform syntax, and concrete edge cases encountered in practice. Lilac abstracts these insights into formal lifting rules that can be automatically applied, verified, and incrementally refined as more training data becomes available.

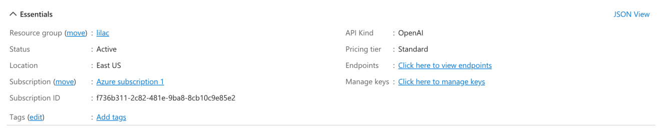
From a system design perspective, our approach relies on iterative engineering workflows and manual orchestration of lifting stages. In contrast, Lilac formalizes these stages into a structured pipeline with explicit correctness guarantees. In this sense, our work can be viewed as the empirical and technical foundation that informs and complements Lilac's broader, automated design.

IV. Cloud State Identification in Lifting

An essential prerequisite for IaC lifting is *cloud state identification*, which refers to discovering existing cloud resources, retrieving their detailed configurations, and converting them into a unified JSON representation suitable for downstream lifting. During this semester, I focus on designing python scripts to automate this process across Azure, GCP, and AWS. Since each cloud platform exposes resources through different abstractions and scopes, cloud state identification requires platform-specific workflows.

A. Azure

Azure provides the most convenient support for cloud state identification due to its unified resource model. Azure organizes resources hierarchically using subscriptions and resource groups. Subscriptions serve as the primary unit for billing and access control, while resource groups act as logical containers for lifecycle management of related resources. The Azure CLI offers official commands such as `az group list` to enumerate all resources groups within a subscription, and `az resource show` to retrieve detailed resource descriptions. These commands return structured JSON outputs with consistent schemas, making them straightforward to parse and process. As a result, Azure's output format is treated as the reference standard for cloud state identification.



Resource group (move)	: lilac	API Kind	: OpenAI
Status	: Active	Pricing tier	: Standard
Location	: East US	Endpoints	: Click here to view endpoints
Subscription (move)	: Azure subscription 1	Manage keys	: Click here to manage keys
Subscription ID	: f736b311-2c82-481e-9ba8-8cb10c9e85e2		
Tags (edit)	: Add tags		

Fig. 1. Lilac Cloud State Identification

The recorded JSON file, shown in Listing 1, captures the detailed configuration of the resource, including properties such as endpoints and capabilities.

Listing 1. Recorded Azure OpenAI Resource

```
{
  "value": [
    {
      "id": "/subscriptions/f736b311.../testingLiLac",
      "name": "testingLiLac",
      "location": "eastus",
      "kind": "OpenAI",
      "properties": {
        "endpoint": "https://testinglilac.openai.azure.com/",
        "publicNetworkAccess": "Enabled",
        "capabilities": [
          { "name": "VirtualNetworks" },
          { "name": "MaxFineTuneCount", "value": "500" }
        ]
      },
      "resourceGroup": "lilac",
      "type": "Microsoft.CognitiveServices/accounts"
    }
  ]
}
```

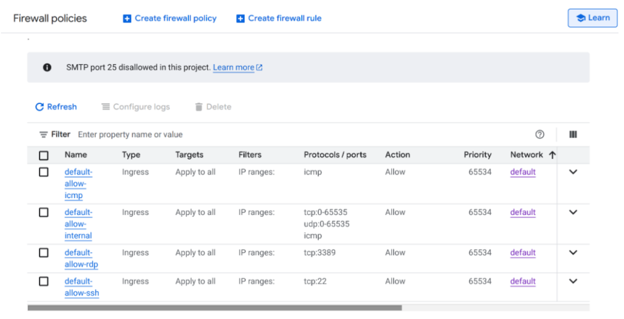
B. GCP

Google Cloud Platform (GCP) adopts a project-centric resource model, where all resources are scoped to projects. Unlike Azure's two-level hierarchy of Subscriptions and Resource Groups, GCP flattens these concepts into Projects, which serve as the singular container for billing, access control, and resource isolation. Furthermore, while the `gcloud` CLI supports listing resources within a project, `"gcloud", "asset", "search-all-resources", "--project", project_id, "--format=json", "--quiet"`, it does not provide a unified command for describing all resource types. Each service exposes service-specific commands. For example, describing a compute instance requires parsing its resource ID to extract the zone and passing it via a `--zone` flag, `"gcloud compute instances describe <instance-id> --zone <zone>"`, whereas describing a subnetwork requires extracting and passing a `--region` flag, `"gcloud compute networks subnets describe <subnetwork-id> --region <region>"`. Even more complex, resources like DNS record sets cannot be described directly by ID; they require an auxiliary lookup to resolve the parent managed zone's ID to a name before the record set can be queried. This lack of a uniform interface compels the identification script to maintain a large set of conditional parsing rules for every supported resource type, contrasting sharply with Azure's consistent model.

To overcome the fragmented command structure in GCP, we integrated an LLM-based resolver using the OpenAI API. This component automates the inference of correct `gcloud` commands for any given resource type. The workflow follows a structured, sequential process:

1) **Input Aggregation**: Resources identified by the initial global search are grouped by their asset type. Items with unknown hardware settings that cannot be resolved by static rules are collected into a batch list.
2) **Prompt Construction**: For each batch, we construct a prompt containing a system instruction acting as a "GCP CLI expert." We provide few-shot examples to guide the model on how to extract zones, regions, and names from complex resource IDs. For instance, the model is shown that a subnetwork ID implies a `--region` flag, while a VM instance ID implies a `--zone` flag.
3) **Batch Inference**: The constructed prompt is sent to the ChatGPT API. We process resources in batches (e.g., 20 items) to optimize network latency and token usage. The model returns a strict JSON array of `gcloud` command arguments corresponding to the input list.
4) **Response Parsing & Caching**: The script parses the returned JSON array. Valid commands are executed immediately to retrieve the resource details. Crucially, the generated command structure is cached by asset type (e.g., mapping `compute.googleapis.com/Instance` to its specific command template). Future occurrences of the same asset type retrieve the template from the cache, avoiding redundant API calls.
5) **Execution**: Finally, the resolved `gcloud` commands are executed against the live GCP environment. The standard JSON output is captured and normalized into our unified schema for downstream processing.

C. Recorded JSON Artifact


Firewall policies

Create firewall policy | Create firewall rule | Learn

SMTP port 25 disallowed in this project. Learn more

Refresh | Configure logs | Delete

Filter: Enter property name or value

Name	Type	Targets	Filters	Protocols / ports	Action	Priority	Network
default-allow-icmp	Ingress	Apply to all	IP ranges:	icmp	Allow	65534	default
default-allow-internal	Ingress	Apply to all	IP ranges:	tcp:0-65535 udp:0-65535 icmp	Allow	65534	default
default-allow-rdp	Ingress	Apply to all	IP ranges:	tcp:3389	Allow	65534	default
default-allow-ssh	Ingress	Apply to all	IP ranges:	tcp:22	Allow	65534	default

Fig. 2. GCP Firewall Policy

```
{
  "value": [
    {
      "allowed": [
        {
          "IPProtocol": "tcp",
          "ports": [ "22" ]
        }
      ],
      "description": "Allow SSH from anywhere",
      ...
      "name": "default-allow-ssh",
      ...
      "priority": 65534,
      "sourceRanges": [ "0.0.0.0/0" ],
      "assetType": "compute.googleapis.com/Firewall"
    },
    {
      "allowed": [
        {
          "IPProtocol": "tcp",
          "ports": [ "0-65535" ]
        },
        ...
      ],
      "description": "Allow internal traffic on the default network",
      ...
      "name": "default-allow-internal",
      ...
    },
    {
      "allowed": [
        {
          "IPProtocol": "tcp",
          "ports": [ "3389" ]
        }
      ],
      "description": "Allow RDP from anywhere",
      ...
      "name": "default-allow-rdp",
      ...
    },
    {
      "allowed": [
        {
          "IPProtocol": "icmp"
        }
      ],
      "description": "Allow ICMP from anywhere",
      ...
      "name": "default-allow-icmp",
      ...
    }
  ]
}
```

D. AWS

AWS architecture shares more DNA with GCP than Azure, primarily due to its service-centric rather than resource-centric API design. Consequently, identifying the cloud state in AWS faces similar fragmentation challenges. While Azure offers a uniform `az resource show`, AWS separates control planes by service, e.g., `aws ec2 describe-instances`, `aws s3api list-buckets`, and `aws rds describe-db-instances`. Although we have not yet implemented the identification script for AWS, we propose a hybrid identification strategy mirroring our GCP approach:

1) **Global Discovery**: We utilize the `aws resourcegroupstaggingapi get-resources` command as a coarse-grained aggregator. This API acts similarly to GCP's Asset Inventory, allowing us to enumerate resources across regions (by iterating through standard regions like `us-east-1`, `us-west-2`) and resource types.
2) **LLM-Driven Command Resolution**: Given the vast number of service-specific namespaces in the AWS CLI, we employ the same ChatGPT-based resolver pattern. The aggregated ARNs (Amazon Resource Names) from the discovery phase serve as inputs. The LLM then constructs the precise `describe` or `get` commands—for example, mapping an ARN like `arn:aws:s3:::my-bucket` to `aws s3api get-bucket-location` and `aws s3api get-bucket-acl`, or an EC2 ARN to `aws ec2 describe-instances --instance-ids <id>`.
3) **Execution & Normalization**: The generated commands are executed against the AWS environment. The resulting heterogenous JSON outputs are then normalized into our unified Azure-like schema, ensuring that despite the underlying API differences, the final state representation remains consistent for IaC processing.

V. Conclusion

We have demonstrated a unified framework for identifying cloud resources across Azure, GCP, and AWS. While Azure's consistent resource model simplifies discovery, the fragmented APIs of GCP and AWS necessitate a hybrid approach combining LLM-powered command resolution. This methodology effectively normalizes distinct cloud states into a single schema for further Lifting processing. Future work will focus on implementing the AWS identification script and organize the codebase into a more maintainable structure.