Diploma Project

TERRAFORM PROVIDER FOR CLOUD-LAB

Panagiotis Grigoriou

University of Cyprus
Department of Computer Science

Department of Computer Science

May 2025

UNIVERSITY OF CYPRUS

Faculty of Pure and Applied Sciences

Department of Computer Science

May 2025

TERRAFORM PROVIDER FOR CLOUD-LAB

Panagiotis Grigoriou

Advisor

Haris Volos

Acknowledgements

I want to sincerely thank Professor Haris Volos, my supervisor, for his advice and introduction to the subject of my thesis. Additionally, I am immensely appreciative of my family and friends' understanding and support during my studies.

ABSTRACT

This thesis presents the design and implementation of a custom Terraform provider for CloudLab, so enabling Infrastructure-as-Code (IaC) paradigms to manage CloudLab resources.The widely used IaC tool HashiCorp Terraform lets users repeatedly declaratively define and provision infrastructure. Developing a CloudLab Terraform provider allows us to bridge the gap between CloudLab's traditional experiment workflow, in which users must manually define, instantiate, monitor, and terminate experiments via the web portal or bespoke scripts, and modern Infrastructure-as-Code tools that support fully declarative, automated provisioning and lifecycle management. The provider is implemented in Go, and uses the API of the CloudLab portal to interact with CloudLab's backend.

Our solution uses Python Flask as an intermediary to invoke current CloudLab portal tools and, when needed, Selenium browser automation to replicate user actions on the CloudLab web interface since several CloudLab operations lack an official REST interface. We describe the architecture of this integrated system in detail, including how Terraform requests are translated into CloudLab experiment lifecycle operations (instantiation, monitoring, termination). The solution is evaluated against the traditional CloudLab web portal usage, demonstrating improved automation, repeatability, and integration potential. We address constraints including those resulting from Terraform's plugin model and CloudLab's design (e.g., experiment elasticity), and we suggest future improvements. This project shows how feasible it is to treat experimental cloud environments as code, so improving reproducibility and simplifying the integration of CloudLab experiments into ongoing development pipelines.

We address constraints including those resulting from Terraform's plugin model and CloudLab's design (e.g., experiment elasticity), and we suggest future directions. This work shows the viability and advantages of treating experimental cloud environments as code, so improving repeatability and simplifying the integration of CloudLab experiments into ongoing development pipelines.

TABLE OF CONTENTS

LIST OF FIGURES

1 Introduction

CloudLab is intended for research and education on cloud architectures [2]. It lets users access configurable low-level servers, networking and storage, enabling ground-up experiments to build whole cloud systems [3]. Beyond what standard public clouds provide, CloudLab stresses control, visibility, and performance isolation to support completely customized and repeatable experiments on dedicated hardware.

In CloudLab, users define profiles that describe the desired experiment topology (machines, network links, disk images, etc.), and an experiment is an instantiation of a profile on available hardware. When a profile is instantiated, CloudLab allocates the specified resources and configures them with the requested software, allowing many instances of the same profile to run concurrently on separate hardware. Standard profiles exist for common cloud stacks (e.g., an OpenStack cluster profile), as well as base operating system environments. Users can also create custom profiles and share them with others to facilitate repeatable research experiments [2, 3].

Despite CloudLab's power and flexibility, interacting with it traditionally requires manual or ad-hoc methods. Researchers typically use the CloudLab web portal to start and manage experiments, or write custom scripts using CloudLab's libraries (such as the geni-lib Python API) to describe and instantiate profiles. The Web portal provides a graphical interface for selecting a profile, configuring parameters, and launching an experiment, but this process means doing multiple manual steps through a browser. For example, to deploy an experiment, a user must log into the portal, navigate to the profile, fill out parameters (if any), and click instantiate, then later terminate the experiment via the User Interface. While effective, this manual approach is error-prone and not easily repeatable in an automated pipeline.

CloudLab offers a programmatic Portal API that allows experiments to be controlled without the GUI [3], but using it directly requires writing custom code (for instance, using portal-specific client libraries) and handling CloudLab's authentication and experiment state logic. On the other hand, Infrastructure-as-Code tools let you specify infrastructure in declarative configuration files, enabling automated, repeatable provisioning, and teardown of complex environments.

HashiCorp Terraform is one of the leading IaC tools that enable developers to write human-readable configuration files to provision, update, and destroy infrastructure components across dozens of cloud services and providers.

Terraform reads infrastructure specifications written in a configuration file and then creates or adjusts the corresponding cloud resources. Its core benefits include version control (infrastructure changes tracked along with application code), automation (one-command deployments), and reproducibility (the same configuration always yields the same environment). Under the hood, Terraform's plugin-based architecture uses providers, modules that speak to the APIs of specific platforms. There are official or community-maintained providers for major clouds (AWS, Azure, Google Cloud, etc.) and many other services, which together give Terraform a unified workflow for managing heterogeneous infrastructure.

1.1 Motivation and problem description

CloudLab, despite being cloud-like in providing compute resources on demand, does not have a Terraform provider or similar IaC integration. This means that researchers could not easily include CloudLab experiments in automated infrastructure deployments or continuous integration pipelines using Terraform. Every CloudLab experiment instantiation is a separate process, decoupled from the IaC-managed parts of an experiment (for instance, setting up cloud resources to interact with a CloudLab experiment would require coordinating the two environments manually).

The lack of IaC integration potentially hampers experiment reproducibility and efficiency - for example, sharing a CloudLab experiment setup currently requires describing the steps or providing a profile script, rather than sharing a ready-to-run Terraform configuration. By developing a custom Terraform provider for CloudLab, we aim to enable CloudLab users to define their experiments in code and deploy or tear them down with simple commands, just as they would for resources in AWS or other clouds. This integration would allow CloudLab experiments to be treated as part of a larger infrastructure description, enabling hybrid workflows (e.g., provisioning a CloudLab experiment and cloud VMs together) and embedding experiment setup into reproducible scripts.

1.2 Aims and Objectives

This thesis was built on a previous implementation that could manage simple CloudLab resources. Specifically, the provider should allow users to declaratively specify CloudLab experiments in a Terraform configuration file. Upon running Terraform, the provider will handle authenticating to CloudLab, instantiating the specified experiment profile, waiting for resources to become ready, and exposing relevant information back to the user (such as experiment IDs or node IPs). The provider should also help destroy experiments using Terraform, so closing the lifetime cycle.

Comparison with the older simple resource

The original `Terraform-profile` provided in the UCY-COAST project allowed [4] users to create single node resources (experiments) with a fixed configuration of Ubuntu 18 and on the Emulab cluster only. In contrast, the provider developed in this thesis offers:

- `cloudlab_simple_experiment`: Launches multi-node experiments with configurable node counts and optional blockstore volumes, all in one Terraform resource. It now supports selecting from multiple Ubuntu images (e.g., 24.04, 22.04, 20.04) and lets users specify which cluster their experiment should run on.
- `cloudlab_openstack_experiment`: Deploys a full OpenStack environment (controller + compute nodes) plus a persistent 100 GB dataset that survives across runs. This profile also allows the user to define the type of node they want specifically (for example, c220g5).

1.3 Contributions

This thesis builds on Panis' implementation [4] of CloudLab infrastructure management with Terraform and introduces the following key improvements.

1. **Unified Code Repository.**

   We have consolidated the functionality that was previously split between two separate repositories into a single repository. This simplifies development, testing, and distribution by ensuring that all provider code and CloudLab profile definitions are maintained together.

2. **Automated Flask API Initialization.**

   The Flask-based intermediary API service now starts automatically in the background upon the first invocation of `terraform apply`, eliminating the manual step required by the user to start the server separately in another window.

3. **Selenium-Powered Portal Automation.**

   Panis' original prototype required users to *manually*

   (a) visit the CloudLab portal to download the encrypted `cloudlab.pem` credential,

   (b) decrypt it locally with `openssl`, and

   (c) copy the resulting X.509 key into the provider repository and the flask repository.

   Because the provider cannot see the web interface, experiments launched outside Terraform would orphan resources and complicate debugging. To eliminate these issues, we embed a lightweight, headless Selenium WebDriver in a side service that:

   - **Zero-touch credential management** Upon initialization, the headless browser logs in, downloads a fresh `cloudlab.pem`, decrypts it, and hands the short-lived key to the provider entirely in memory, removing all manual OpenSSL steps.
   - **Live state synchronization** Selenium periodically scrapes the *My Experiments* dashboard (experiment ID, profile, status, expiry) to get the latest state of experiments.

4. **CloudLab VM Resource - Multi-Node Support and Additional Storage.**

   We upgraded the `cloudlab_vm` Terraform resource to support:

   - deployment of multiple nodes within a single CloudLab experiment, allowing users to provision small clusters in one step;
   - attachment of an additional blockstore volume (e.g. 100 GB) to the experiment, using an upgraded parameterized profile (https://github.com/pgrigo01/storage-profile).

5. **New CloudLab OpenStack Resource with Shared Storage.**

   We added a `cloudlab_openstack_experiment` resource that leverages an OpenStack profile [5]. This resource automatically deploys a multi-node OpenStack cluster and provisions a shared 100 GB persistent dataset that survives across experiment runs.

6. **Advanced CloudLab Profile Integration.**

   The previous implementation supported only single-node experiments on the Emulab Cluster and parametrization was not really possible. Unlike the basic single-node profile used before, our solution supports parameterized topologies and persistent storage. By embedding these rich profiles directly into the provider workflow, users can define complex bare-metal clusters or private OpenStack clouds, complete with persistent volumes-purely in Terraform.

1.4 Structure of the Thesis

The remainder of this thesis is organized as follows. In Chapter 2, we cover the background and methodology, explaining the key technologies and concepts underlying our work. Infrastructure-as-Code and Terraform, the CloudLab platform, Terraform providers, the Flask framework, and the Selenium automation tool. Chapter 3 presents how the user can setup and use the Terraform Provider, it also demonstrates how resources need to be defined to create experiments in the configuration file. Chapter 4 presents the design and implementation of the custom CloudLab Terraform provider, describing the design components of the system and their interactions. We include detailed explanations of the provider implementation, the Flask-based API service that bridges to CloudLab, and how Selenium is utilized. In Chapter 5, we compare and discuss the Terraform-based approach with the Cloudlab website. Chapter 6 discusses the limitations and challenges encountered, including limitations inherent in CloudLab, limitations of Terraform's plugin mechanism, and issues faced in the Go development process. Finally, Chapter 7 concludes the thesis, summarizing the achievements and suggesting future additions and improvements.

2 Background

In this chapter, we introduce the fundamental ideas and tools that support our work. Our project spans the domains of cloud infrastructure management and web automation; thus, we cover the relevant background on Infrastructure as Code (Terraform in particular), the CloudLab testbed's operating model, Terraform's provider plugin system, and the tools (Flask and Selenium) applied in our implementation. Appreciating the design decisions and contributions of this thesis depends on the knowledge of these ideas. We also describe how we merged several technologies to reach the intended integration.

2.1 CloudLab Testbed Overview

CloudLab is a large-scale federated testbed for cloud computing research. It was launched in 2014 through NSF support, and it operates across multiple sites (University of Utah, University of Wisconsin, Clemson University, etc.) with over a thousand machines in total. CloudLab's mission is to enable researchers to experiment with new cloud architectures by giving them as much control as possible over the underlying hardware. In CloudLab, users can request bare metal servers (or virtual machines on those servers) and networking equipment with minimal interference from the platform, thereby allowing low-level experimentation (for example, custom virtualization stacks, novel storage systems, or network experiments that are not possible on public clouds). Over its years of operation, CloudLab has served thousands of users and tens of thousands of experiments, becoming a key facility for reproducible research in systems and networking.

CloudLab inherits much of its design from earlier testbeds like Emulab and GENI. The fundamental abstraction in CloudLab is the profile (sometimes called an RSpec in GENI terminology). A profile is essentially a declarative specification of an experiment's topology: it lists the compute nodes, their hardware types, the disk images or OS to load on them, network link definitions,

and any experiment-specific scripts to run on startup. CloudLab profiles can be created through a graphical profile builder on the web portal, by writing an RSpec XML, or by using a library like geni-lib (Python) to script the definition. Once a profile is defined, it acts as a template or blueprint. Launching an experiment in CloudLab means instantiating a profile. As the CloudLab team describes: “Experiments in CloudLab are instances of profiles. A profile contains a description of the hardware resources (servers, switches, etc.) that the experiment will run on, and the software needed to run the experiment. When a profile is instantiated, CloudLab selects the available hardware that matches the profile specification and provisions that hardware with the software and configuration described in the profile. Each experiment is isolated on its allocated resources, and multiple experiments (even from the same profile) can run simultaneously on different hardware slices. Below we can see how an example profile that supports a single node with the options : This minimal GENI/CloudLab profile lets you pick an aggregate site and then launches exactly one XenVM running Ubuntu 24.04. It defines two parameters (the aggregate and the fixed Ubuntu 24.04 image), binds and verifies them, builds the RSpec, creates a VM named ’node’ on your chosen aggregate, and creates the node.

```
import geni.portal as portal
import geni.rspec.igext as ig
pc = portal.Context()
agglist = [
    ("urn:publicid:IDN+utah.cloudlab.us+authority+cm","utah.cloudlab.us"),
    ("urn:publicid:IDN+wisc.cloudlab.us+authority+cm","wisc.cloudlab.us")]
imagelist = [
    ('urn:publicid:IDN+emulab.net+image+emulab-ops//UBUNTU24-64-STD', 'Ubuntu
        24.04'),
]
# Define parameters
pc.defineParameter(
    "aggregate", "Specific Aggregate",
    portal.ParameterType.STRING,
    agglist[0][0], agglist)
pc.defineParameter(
    "image", "Node Image",
    portal.ParameterType.IMAGE,
    imagelist[0][0], imagelist,)
# Bind & verify
params = pc.bindParameters()
pc.verifyParameters()
# Build the RSpec
request = pc.makeRequestRSpec()
# Single VM, with chosen image & aggregate
node = ig.XenVM("node")
node.disk_image = params.image
if params.aggregate:
    node.component_manager_id = params.aggregate
request.addResource(node)
pc.printRequestRSpec(request)
```

Listing 2.1: `exampleProfile`

CloudLab provides a library of standard profiles for common use cases. For example, there is a standard OpenStack profile that will set up an OpenStack deployment on a handful of nodes (one controller, several compute nodes) – this helps researchers to quickly get a private cloud environment for experiments (this is also now implemented. Other standard profiles include Hadoop/Spark clusters, Kubernetes clusters, bare Ubuntu installations, and more. Users are

also free to create their own profiles, which can include custom OS images or startup scripts for specialized experiments. Profiles can be shared publicly or within a project team, promoting collaboration and repeatability (these profiles van be defined using a github repository as shown in the image below). CloudLab also supports versioning of profiles – changes to a profile can be saved as new versions, and old versions can be instantiated for archival experiments, enhancing reproducibility of past results. The two supported profiles now for the Terraform provider are the Openstack profile and the terraform-storage profile.

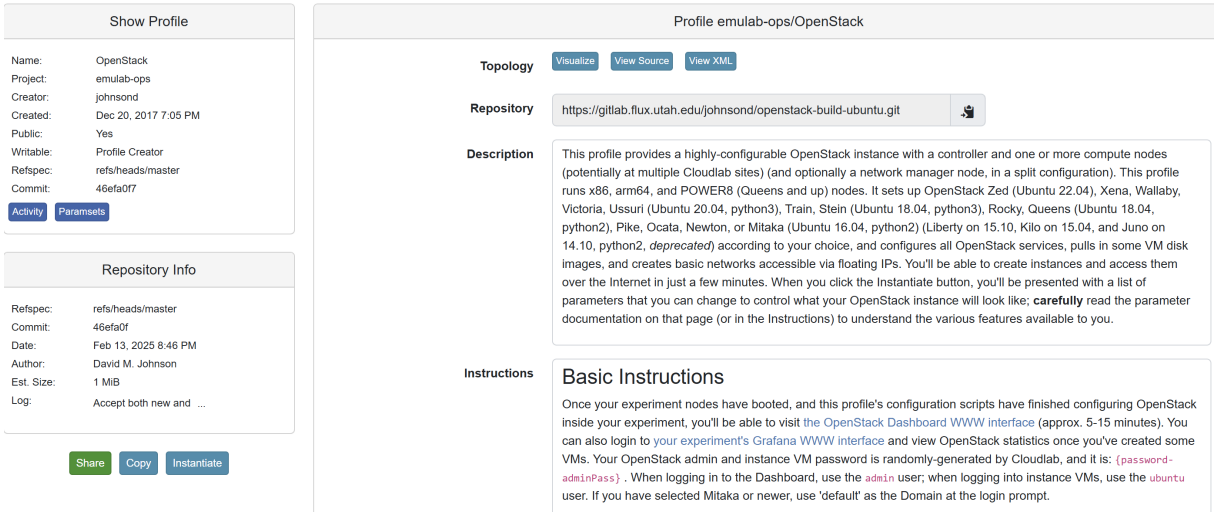

Figure 2.1: Openstack Profile

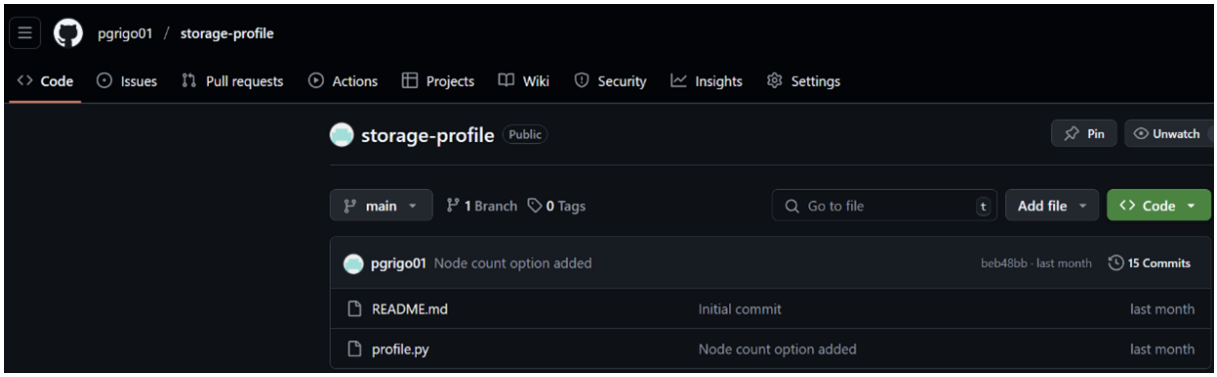

Figure 2.2: terraform-storage Profile

A notable design aspect of CloudLab is its focus on complete environments-by which we mean defining and launching your entire experiment (every node, network link, storage volume, and configuration) as a single, fixed topology-rather than dynamic scaling of individual resources. In a public cloud (like AWS), one might start with no servers running, then programmatically create a VM or scale out additional VMs on demand, and terminate them when done. CloudLab instead encourages users to define a fixed experiment topology (which could be one node or

many) and instantiate it as a whole. Experiments have a maximum duration (by default 16 hours, but this can be extended), after which resources are freed. Once your experiment is instantiated, its topology is fixed: if you realize you need more or fewer nodes, you must edit the profile to add or remove node definitions and then re-provision the experiment-either by terminating and starting a new run or by using the Portal's "modify" action, which under the hood tears down and rebuilds the environment to match the updated profile. This means CloudLab is less elastic in the cloud sense, but this trade-off exists to describe a complete, repeatable environment, making it easier to repeat experiments in a consistent setting. As noted by the CloudLab designers, the emphasis on reproducibility over elasticity can initially confuse new users, but ultimately it provides an easier way to get a known environment for each experiment run.

Users interact with CloudLab primarily through the CloudLab Portal, a web-based interface. Through this portal, users can create or join projects, define profiles (via an editor or by uploading profile descriptions), and instantiate or terminate experiments. The portal provides status information on experiments and offers web-based consoles or SSH key management to log into nodes. In addition to the GUI, CloudLab experiments can be controlled via command-line tools or APIs inherited from GENI. For instance, advanced users can use the Portal API which CloudLab exposes for programmatic experiment management. This API uses a user's downloaded certificate (cloudlab.pem that is downloaded from the web interface Figure 2.3) and allows operations like creating an experiment from a profile, polling its status, and terminating it. However, this API is not widely advertised for everyday users and typically requires writing custom code (in Python or another language) to use. The portal documentation shows examples of how to the API with scripting. In summary, before our work, if a researcher wanted to automate CloudLab experiment deployment as part of a larger workflow, they would need to manually script against CloudLab's API or CLI tools, which is a bespoke solution in each case.

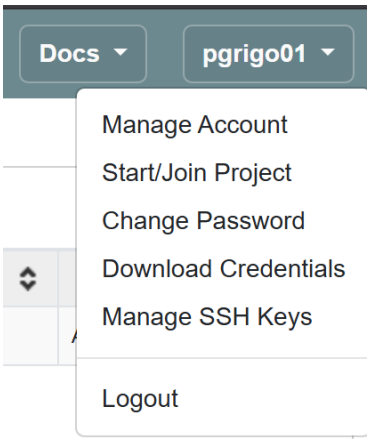

Figure 2.3: Credentials file

This context sets the stage for why integrating CloudLab with Terraform is beneficial. CloudLab already embodies the idea of describing infrastructure as code to some extent (through profiles), but it does not integrate with the broader ecosystem of IaC tools. By bringing CloudLab into Terraform, we aim to make CloudLab experiments first-class elements of reproducible infrastructure pipelines. This would unite CloudLab's powerful bare-metal provisioning with Terraform's automation and multi-cloud workflow. In the next sections, we discuss how Terraform can be extended to new platforms, via providers, and the specific tools we used to implement the CloudLab provider.

2.2 Infrastructure as Code and Terraform

Infrastructure as Code (IaC)[6] in IT infrastructure management that promotes writing and executing code to define, provision, and configure infrastructure resources(experiments). IaC lets engineers indicate the desired state of infrastructure in declarative configuration files or scripts rather than manually clicking through cloud consoles or running imperative commands for every change. The IaC tool then either automatically generates or modifies the actual resources to fit the stated condition. This method brings infrastructure management software engineering techniques, so enhancing repeatability and lowering human error. Treating infrastructure definition as code will enable version control, review, and testing of the same nature as application code. Since the same configuration can be used many times to produce identical results, IaC generates more consistent and repeatable environments. Building an environment from code is far faster and less prone to omission than doing the steps manually; hence, it also promotes faster recovery and scaling.

Among the most widely used IaC applications available is Terraform [7]. Using a high-level, human-readable configuration language (HCL), Terraform lets users specify both cloud and on-site resources (servers, networks, DNS records, etc.). Terraform designs are declarative, stating what infrastructure is intended, rather than how to get it. The Terraform engine takes these configurations and performs a planning phase to calculate the changes needed to reach the desired state, then executes those changes to provision or modify resources accordingly. Terraform is an open-source tool that has become a standard for multi-cloud infrastructure automation.

When a user wants to add or remove an experiment, they edit the Terraform configuration file (e.g., main.tf) and, if it is the first time, run terraform init to download the CloudLab provider. After each change, they can run ‘terraform plan‘ to preview infrastructure changes and then terraform apply to create or destroy the specified experiments.

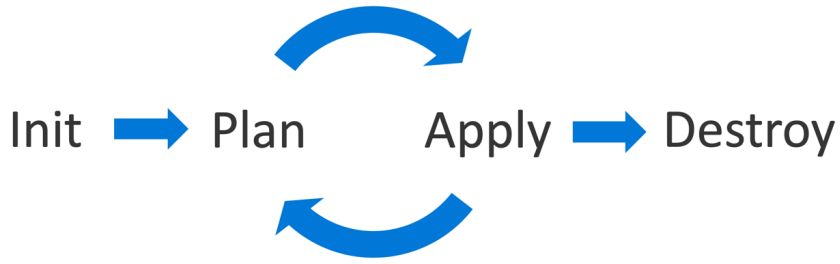

Figure 2.4: Terraform Cycle

One of Terraform's strong points is its capacity to handle complex dependencies among resources. Based on configurable references, the tool automatically determines the proper sequence to create or destroy resources. It also keeps a state file to recall which resource it produced (terraform.tfstate) [8]. Usually housed in source repositories, Terraform configurations allow teams to develop infrastructure cooperatively and conduct peer reviews of infrastructure modifications. The result is reliable, traceable infrastructure deployments: teams can recreate entire environments (for development, testing, production for experiments) with one command and be confident that each run produces the same outcome. In scientific research, where infrastructure variations can influence outcomes, this repeatability is especially important; IaC guarantees a consistent environment for an experiment every time it is supplied. Note that while Terraform and similar IaC tools excel at deploying resources, they typically focus on provisioning infrastructure rather than managing the software inside those resources. In the context of CloudLab the distinction is blurred: a CloudLab profile often includes both the hardware specification and the software environment. Thus, using Terraform for CloudLab means that we treat an entire experiment (hardware + software configuration) as a single unit of infrastructure to be provisioned.

2.3 Terraform Providers Concept

Terraform's core is provider-agnostic: it knows how to orchestrate resource creation and manage state, but relies on provider plugins to actually talk to the RESTful or RPC interfaces exposed by cloud service providers (e.g. AWS, Azure, Google Cloud), SaaS platforms, or other infrastructure systems. A Terraform provider is a plugin (typically implemented in Go) [8] that informs Terraform of the resource types and data sources it can manage, and contains the logic to perform CRUD (Create, Read, Update, Delete) operations on those resources using the target platform's API. In essence, providers are the translation layer between Terraform's desired state and the real-world API calls needed to achieve that state. A provider knows how to create each type of resource by calling the appropriate API endpoints with the correct parameters, how to update or

delete those resources, and how to read their current state.

A Terraform provider typically defines one or more resource types. For example, the AWS provider has resource types like aws_instance[9], aws_s3_bucket [10]. Below is a minimal Terraform configuration that demonstrates how to declare the AWS provider (with an explicit profile and region) and then use two common AWS resource types, an EC2 instance and an S3 bucket.

```
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "~> 4.16"}}
  required_version = ">= 1.2.0"}
provider "aws" {
  profile = "default"
  region = "us-west-2"
}
resource "aws_instance" "web_server" {
  ami          = "ami-830c94e3"
  instance_type = "t2.micro"
  tags = {
    Name = "ExampleWebServer"}}
resource "aws_s3_bucket" "storage" {
  bucket = "bucket-name"
  acl    = "private"
}
```

Listing 2.2: Example Amanzon configuration

Each resource type has a schema (defining what configuration attributes it accepts and what attributes it exports). The user writes Terraform configurations using these resource types. When terraform apply is run, Terraform will load the provider, and for each resource instance in the configuration, it will invoke the provider implementation. The provider must implement functions to handle: creating the resource, reading the resource (to refresh Terraform state with the actual live data, or to validate it exists), updating it (if possible, given differences in configuration) and deleting it. In many cases, not all operations are needed if a resource cannot be updated in place, the provider can document that it forces a new resource (Terraform will then call create+delete instead of update when configuration changes). The provider can also define data sources (to fetch data from the API without creating a resource) and provider-level config-

uration (like region, credentials, endpoints). The official Terraform documentation outlines best practices for developing custom providers.

Under the hood, during an apply Terraform Core loads the CloudLab provider plugin (our Go code) and parses the user's configuration (including things like the CloudLab certificate file). Terraform compares the desired state against the current state stored in 'terraform.tfstate', determines which experiments need to be created, updated, or destroyed, and invokes the matching provider. We can see below the general architecture of Terraform.

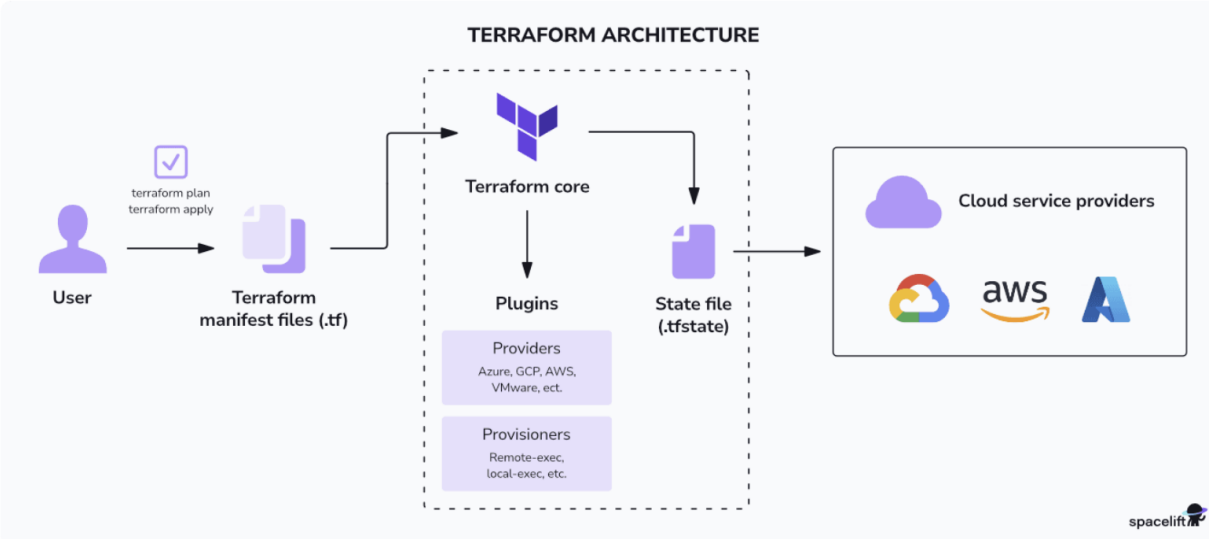

Figure 2.5: Terraform Architecture [1]

2.4 Flask Web Framework

Flask is a lightweight web application framework for Python. It is called a micro-framework since it offers the foundations for web server capability free from strict project structure requirements or heavy dependencies. Built to grow to complex applications, it starts quickly and easily. With minimum boilerplate, Flask provides a simple interface for Python handling HTTP requests and route definition (URL endpoints), so enabling the development of RESTful APIs or web services. Its minimalist core can be extended with numerous community extensions for things like authentication, database integration, etc., as needed.

We chose Flask as part of our methodology to serve as an integration layer between Terraform (which runs our Go provider plugin) and CloudLab's control interface (which, as discussed, might involve complex interactions like certificate-based calls or web automation best done in Python). The idea was to create a small Flask web service that runs locally (on the same machine as Terraform) and exposes a few HTTP endpoints corresponding to the operations we need to perform on CloudLab. For example, an endpoint like /create-experiment could accept details

such as the profile ID (or name), project name, and experiment parameters, then perform the sequence of actions to instantiate that experiment on CloudLab. The Terraform provider, instead of containing all that logic in Go, could simply make an HTTP request to the Flask service's endpoint and get a response indicating success or failure (and any data like the experiment's unique identifier).

Python already has tools and libraries (including some CloudLab/GENI client libraries) as well as portal-tool capability to manage CloudLab interactions. By using Flask, we could write those interactions in Python and avoid reinventing them in Go. Loading a user's certificate and key to authenticate with CloudLab's XML-RPC API, for example, is not possible with Golang current libraries like x509 library or other third party ones . All Flask does is act as a wrapper to expose those Python capabilities across HTTP.

Flask's simplicity meant we could implement this service quickly. For designing experiments, monitoring experiment status, deleting experiments, etc., we just needed a few endpoints. Flask manages all the HTTP server details; hence, our code could concentrate on the logic particular to CloudLab. This was far faster than, say, building a more heavyweight framework or attempting to have the Go plugin elegantly manage multi-step processes.

Running a separate Flask service means that the possibly long-running or blocking operations-like waiting for an experiment to be ready, or Selenium-based web browser control-occur outside the Terraform provider process. Terraform expects providers to run operations and return; thus, it may be problematic if a provider blocks for too long without providing feedback. Our Go-based Terraform provider never has to sit and wait by running long-running chores in a different Flask process. Rather, the provider just tells Flask to start an action and then polls or checks back later for the outcome-Flask manages the blocking work (akin to waiting for an experiment or driving Selenium) and notifies when it is done. This decoupling lets us more precisely manage difficult processes and timeouts and keeps Terraform responsive.

Basically, Flask allows us to use Python's already existing libraries for all the CloudLab integration, expose a basic REST interface for our Go plugin, and keep each component concentrated on its own work between Terraform and CloudLab. The Flask endpoints and their functions in the chapter on architecture are fully broken out here.

2.5 Selenium for Web Automation

Selenium is an open source framework for automating web browsers. It is widely used for browser-based testing of web applications, allowing scripts to control a web browser programmatically to simulate user interactions.[11] The Selenium project provides a WebDriver API that can control popular browsers (Chrome, Firefox, etc.) either on-screen or in headless mode. While primarily intended for testing, Selenium is also sometimes used for web scraping or automating routine web tasks, especially when no API is available for a particular web interface.

2.6 CloudLab Portal Tools API

CloudLab provides a Python-based *portal-tools* suite that wraps its underlying GENI/Emulab Portal API, giving users a simple command-line and library interface for programmatic experiment management [12]. Maintained in a public GitLab repository, this suite includes functions for:

- `startExperiment`: Allocate bare metal or virtual machine resources according to a profile, authenticating with the user's X.509 certificate.
- `experimentStatus`: Poll the XML-RPC API for the current state (Pending, Active, Failed) of one or more experiments.
- `terminateExperiment`: Cleanly tear down and release all resources associated with an experiment.

Each of these tools communicates directly with CloudLab's XML-RPC endpoints over HTTPS, leveraging the user's downloaded `.pem` certificate and private key to perform mutual TLS authentication [13]. By wrapping low-level Portal API calls in well-tested Python scripts, *portal-tools* eliminates the need to craft raw RPC messages or manage certificate handshakes manually [3].

Our solution embeds these *portal-tools* commands in a lightweight Flask microservice. Whenever Terraform needs to create, check, or destroy an experiment, the Go provider issues a local REST call to the Flask server, which in turn invokes the appropriate *portal-tools* function and captures its JSON output. This decoupling keeps the provider code simple-Terraform only deals with standard HTTP/JSON-while all certificate management, retry logic, and experiment-state handling remains in Python, reusing CloudLab's own tooling [14].

3 Usage

In this chapter, we guide the reader through the end-to-end process of installing, configuring, and using the Terraform CloudLab provider. We begin by describing all prerequisites and running the automated setup script to retrieve credentials and install dependencies. Next, we demonstrate how to initialize the Terraform working directory, configure the provider, and execute the standard Terraform workflow (init, plan, apply) to provision CloudLab experiments. Finally, we provide examples of defining both simple and more complex resources, illustrating how you can leverage the HCL iteration features to scale out multiple experiments efficiently.

3.1 Prerequisites and Setup

Environment Requirements:

- **Go :** Go 1.23+ is required for building the provider
- **Python Dependencies:** Flask, cryptography, APScheduler, and CloudLab XML-RPC packages
- **A browser:** Chrome or Firefox(by default the Chrome is the one that is being installed through the setup script shown below).
- **Terraform**

First clone the github repository: **git clone https://github.com/pgrigo01/terraform-provider-cloudlab**

1) Credential Retrieval and Installations Needed: In order to get the credentials file (cloudlab.pem) the user has to run the provided script setupOnUbuntu.sh that installs go version 1.23, the latest version of terraform , creates a python virtual environment to install all dependencies needed in order to run the flask server in the background and finally prompts the user to give their password and username of Cloudlab to download the credentials file from Cloudlab decrypt it and save it with the name **cloudlab-decrypted.pem** in our workspace.

2) Run: source ./setupOnUbuntu.sh After the **cloudlab-decrypted.pem** appears in our workspace, we can go on to the next step needed.

3) Run source myenv/bin/activate The next step is to enter the virtual environment that was created by the setup script mentioned above.If everything goes well, **(myenv)** must appear in our terminal as shown below.

```
pg@LAPTOP-B27VJ0EA:~/terraform-provider-cloudlab$ source myenv/bin/activate
(myenv) pg@LAPTOP-B27VJ0EA:~/terraform-provider-cloudlab$
```

Figure 3.1: Python Environment Setup

Also the following files will be created after running the setupOnUbuntu.sh encryption_key.key and credentials.encrypted that contain the users encrypted credentials that are used when the flask server starts in the background to poll the current state of our infastructure from Cloudlab.This is helpful for the user since he can now view the experiments that were created from the Cloudlab website, not only the ones that will be created through Terraform.This state will be stored in this file cloudlab_experiments.csv. This is especially useful if the user wants to view what experiments are already created in order to avoid re-declaring the resource for those experiments in the configuration file.Our main.tf file already defines our provider as shown in Listing 3.4 so we are ready to run the next command.

4) The next step is to configure the CloudLab provider in your Terraform configuration:

```
terraform {
  required_providers {
    cloudlab = {
      source = "pgrigo01/cloudlab"
      version = "1.0.0"
    }
  }
}
provider "cloudlab" {
  project        = "UCY-CS499-DC"
  credentials_path = "cloudlab-decrypted.pem"
  browser        = "chrome" # or "firefox"
}
```

Listing 3.1: Provider Configuration

5) The next command to run is **terraform init**, which performs the following actions:

Initializes the working directory

- Creates a `.terraform/` directory to hold local state, plugin binaries, and downloaded modules.
- Writes a lock file (`.terraform.lock.hcl`) to pin provider versions.

Configures the backend

- Reads any back-end block in your configuration and sets up where your Terraform state file will be stored and retrieved.

Downloads provider plugins

- Examines the `required_providers` in your configuration.
- Fetches the matching provider binaries from the Terraform Registry (https://registry.terraform.io) or any other specified source.

Retrieves modules

- Finds all `module` blocks in your configuration.
- Downloads module source code, whether from the Terraform Registry, a GitHub repository, a local path, or other supported destinations, into the `.terraform/modules` folder.

```
(myenv) pg@LAPTOP-B27VJ0EA:~/terraform-provider-cloudlab$ terraform init
Initializing the backend...
Initializing provider plugins...
- Reusing previous version of pgrigo01/cloudlab from the dependency lock file
- Installing pgrigo01/cloudlab v1.0.0...
- Installed pgrigo01/cloudlab v1.0.0 (unauthenticated)

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
(myenv) pg@LAPTOP-B27VJ0EA:~/terraform-provider-cloudlab$
```

Figure 3.2: terraform init Command

Note: If for any reason terraform init fails to bring the terraform plugin you can build it locally by running this script with this command **./build_provider.sh**

6) The next step is to define the resource we want to add.

```
terraform {
  required_providers {
    cloudlab = {
      source = "pgrigo01/cloudlab"
      version = "1.0.0"
    }
  }
}

provider "cloudlab" {
  project        = "UCY-CS499-DC"
  credentials_path = "cloudlab-decrypted.pem"
  browser        = "chrome"
}

resource "cloudlab_simple_experiment" "experiment1" {
  name       = "exp1"
  routable_ip = true
  image      = "UBUNTU 22.04"
  aggregate  = "emulab.net"
  node_count = 1 # optional if not defined defaults to 1
  #extra_disk_space = 50 #optional if not defined no extra space is allocated
}

#Below is an example of an Openstack resource,since it is commented terraform
#will not consider adding it to our infrastructure as we will see below
#only the exp1 will be added

# resource cloudlab_openstack_experiment "experiment2" {
#   name = "exp2"
#   release ="zed"
#   compute_node_count = 1
#   os_node_type = "c220g1"
#   ml2plugin = "openvswitch"
# }
```

Listing 3.2: Example Terraform configuration

7)After declaring which resources we want to add (in the example above in Listing 3.1 were we added one resource) the next command to run is **terraform plan**, which performs the following actions:

Reads configuration and state

- Firstly it reads the configuration file in your working directory(main.tf).
- Loads the current state from the local terraform.tfstate file.

Refreshes resource state

- Queries each resource from to get the real-world state of existing resources(to see if an experiment is still running or if it is terminated to restart it).

Determines resource actions

- Compares the desired configuration against the refreshed state.
- Plans which resources will be created, updated, or destroyed to achieve the target configuration.

Outputs the execution plan

- Prints a human-readable summary of proposed changes, color-coded for create (+), update ( ), and destroy (–).

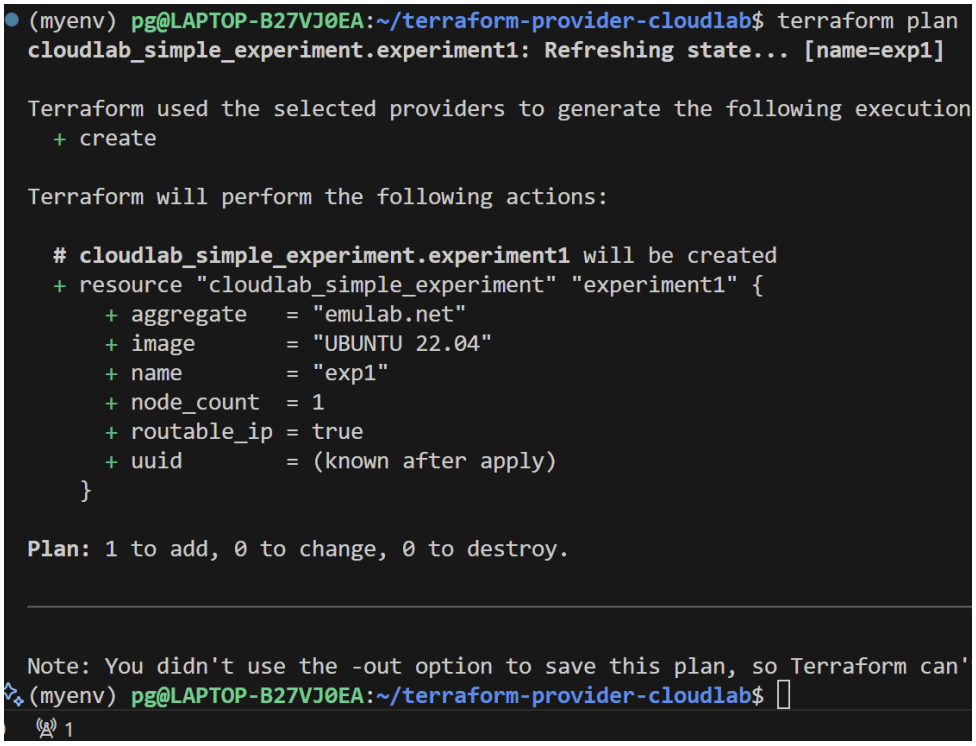

Figure 3.3: terraform plan Run

8) Finally we run **terraform apply**, which performs the following actions:

Reads configuration and state

Refreshes resource state

Obtains or generates an execution plan

Prompts for approval

- Displays the plan summary (create +, update ~, destroy –).
- Pauses and waits for you to confirm (type 'yes').

Executes resource actions

- Creates new resources in your provider.
- Updates existing resources that need modification.
- Destroys resources that were commented out/removed from the .tf file.

Updates state and outputs the results

- Writes the updated state back to `terraform.tfstate`.
- Shows detailed logs of each step, showing attribute values before/after.
- Prints a final output summary.

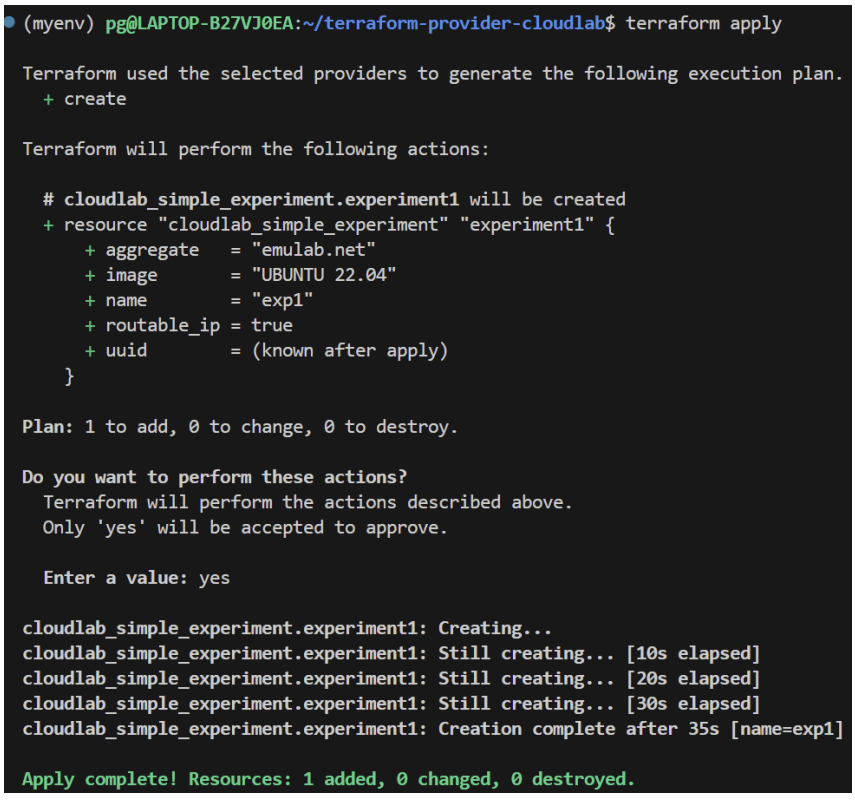

Figure 3.4: terraform apply Run

3.2 Overview of our CloudLab Resources

Before moving to the last step, we need to explain and show what is currently being offered resource-wise.

Resource Schemas: We have two main resource types lets deep dive and see what is currently offered : `cloudlab_simple_experiment` and `cloudlab_openstack_experiment`. These are essentially presets for common profile types. For example, a `cloudlab_openstack_experiment` resource internally uses a known OpenStack profile from Cloudlab.

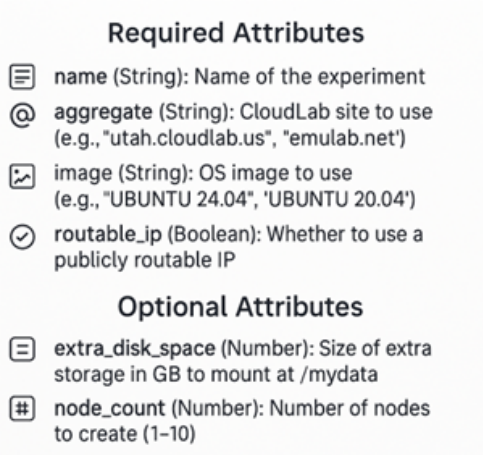

Figure 3.5: cloudlab simple experiment Resource

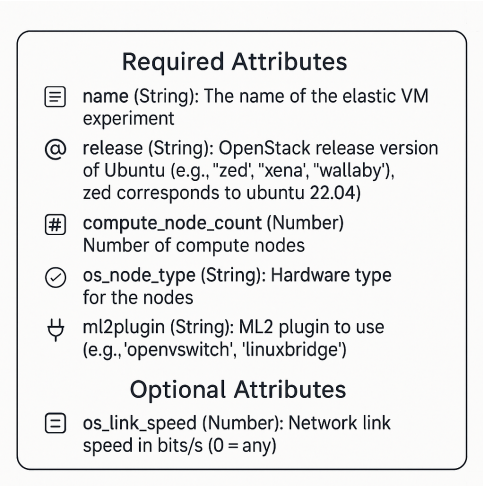

Figure 3.6: cloudlab_openstack_experiment Resource

3.2.1 cloudlab_simple_experiment Resource (cloudlab_vm.go)

- **Schema Definition:** The resource schema has the following attributes: name, aggregate, image, routable_ip, and optional fields extra_disk_space, node_count, and nested vlans.
- **Configure method** During provider configuration, the resource receives a pre-initialized Client instance, giving access to the credentials path, project ID, and browser selection. This allows subsequent API calls to be routed through the local Flask service.
- **Create method:** Reads plan values into a cloudlabvmModel struct. Builds a parameter map, (e.g. UBUNTU 20.04) into URNs using helper functions AddImageParam and AddAggregateParam. Sends a **POST** to /experiment via Client.startExperiment. And when its ready it updates the state.
- **Read method:** On terraform refresh or during subsequent applies, Read verifies the experiment still exists by name; if missing, it removes the resource from state.
- **Delete method:** When a resource is commented out or when terraform destroy is run , then the Delete method calls Client.terminateExperiment. This resource demonstrates a complete mapping between Terraform's declarative configuration and CloudLab's experiment lifecycle, leveraging the Flask intermediary for all portal interactions.

3.2.2 cloudlab_openstack_experiment Resource (elastic_vm.go)

- **Schema Definition:** The resource schema has the following attributes, release, compute_node_count, os_node_type, os_link_speed,ml2plugin.
- **Configure method:** In Configure, the resource marks the shared Client as elastic = true, causing subsequent calls to select the OpenStack profile UUID rather than the default simple profile.
- **Create method:** Reads the plan into an elasticVMModel struct. Constructs a parameter map that matches the OpenStack profile. Sends a POST via Client.startExperiment, uploading the user's .pem and parameters as multipart form data.

  Polls status via Client.experimentStatus until the experiment appears in CloudLab's dashboard,then sets the returned UUID in state.
- **Read method:** Similar to the simple resource, Read confirms the experiment's existence and preserves state if still active; otherwise, it prunes the resource.
- **Update method:** In-place updates are not supported for this resource; any change forces recreation, consistent with Terraform's handling of immutable resources.
- **Delete method:** The Delete method delegates to Client.terminateExperiment, ensuring the elastic experiment is torn down and state is cleaned up.

3.2.3 CloudLab Profiles and Resources

A profile in CloudLab includes all the information about a cloud environment, such as the software that is needed to run it and a description of the hardware (including the network layout) that the software stack should run on.

Making a new profile means making a new geni-lib Python script and, most of the time, one or more disc pictures that go with that script. If someone uses your profile, they will start their own experiment that uses the resources (virtual or real) that are explained in the profile script.

There are two ways to give CloudLab the geni-lib Python script when you make a new profile. First, you can upload it as a single file and let CloudLab handle its storage. Then, when you want to make a new version, you can upload a changed version or edit it in the CloudLab code editor. Secondly, you can use CloudLab's repository-based profile tool. To do this, you need to provide it a git repository URL that has the geni-lib Python script as a profile.py file in the repository's root directory. This lets you control versions based on how you use Git and create new projects from branches.

The easiest way to create a new profile is by copying an existing one and customizing it to your needs. Your goal should be to choose the profile that is most similar to the one you want to build. Usually, this will be one of our facility-provided profiles with a generic installation of Linux.

We implemented two main resource types in Terraform -cloudlab_simple_experiment and cloudlab_openstack_experiment - each backed by a different CloudLab profile. These profiles define the topology and capabilities of the CloudLab experiment that gets created.The figure below illustrates the relationship between our resources and their profiles.

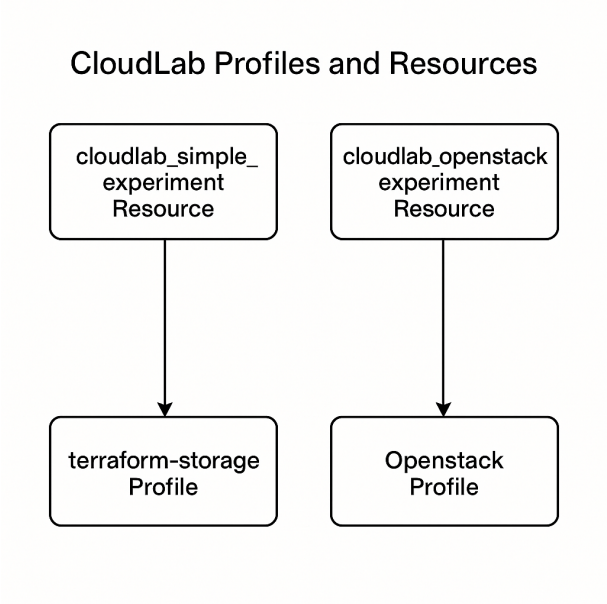

Figure 3.7: Cloudlab Profiles And Resources

cloudlab_simple_experiment Resource and terraform-storage Profile

When a user defines a cloudlab_simple_experiment resource in Terraform with our provider, they can specify how many nodes they want (via a parameter in our configuration named node_count) and optionally request extra disk space. This profile dynamically generates an RSpec (CloudLab experiment description) that includes the requested number of nodes (each loaded with a base OS image) and attaches a blockstore of the given size to the experiment. A blockstore is essentially a chunk of temporary disk provisioned (until the experiment terminates) from CloudLab's storage pool, which CloudLab mounts to one of the nodes (the extra storage is a shared volume mounted on the number of nodes within the same experiment).

The nodes within the experiment are connected on an isolated LAN (so they can communicate). Terraform will treat this whole experiment as one resource. Importantly, all the nodes are created in one go as part of one experiment, which means they can talk to each other with low latency (same LAN) and the experiment can be managed (started or terminated) as a single unit. Instead of manually creating and networking separate single-node cloudlab_vm resources, which was inefficient and spawned a new experiment for each node. Terraform's multi-node profile launches all nodes together in the same experiment, making distributed or master-worker setups much simpler.All things considered, the cloudlab_simple_experiment with the storage-profile offers a flexible cluster right out from Terraform.

Simple Experiment Attributes

Attribute	Required	Description
`name`	Yes	Name for your CloudLab experiment
`routable_ip`	Yes	Whether to assign a routable IP address (boolean)
`image`	Yes	OS image (e.g., "UBUNTU 22.04", "UBUNTU 24.04", "CENTOS 7")
`aggregate`	Yes	CloudLab site to use (e.g., "emulab.net", "utah.cloudlab.us")
`node_count`	No	Number of nodes to create (1-10, default: 1)
`extra_disk_space`	No	Size of extra storage to mount at /mydata in GB
[0] vlans	No	List of VLAN configurations

Available Image Options:

- "UBUNTU 20.04"
- "UBUNTU 22.04"
- "UBUNTU 24.04"
- "CENTOS 7"
- "FreeBSD 11.3"

Available Aggregate Options:

- "emulab.net"
- "utah.cloudlab.us"
- "clemson.cloudlab.us"
- "wisc.cloudlab.us"
- "apt.emulab.net"
- "Any"

Figure 3.8: cloudalb_simple_experiment Attributes

cloudlab_openstack_experiment Resource and Openstack Profile

The cloudlab_openstack resource is designed for a higher-level use case: deploying a ready-to-use OpenStack environment on CloudLab. When this resource is applied, the provider signals the Flask API to launch an experiment using the OpenStack profile. This profile sets up a controller node (which runs OpenStack services) and a number of compute nodes, creating an OpenStack cloud on the allocated hardware. All the nodes are connected via the necessary networks, typically an OpenStack management network and possibly a data network.

By default, this profile includes 100 GB of persistent shared storage volume. This is implemented using the persistent dataset feature of Cloudlab: the profile requests a dataset of 100 GB that is shared among the nodes. This persistent storage is shared between all experiments launched with that profile, meaning that if the experiment is terminated, CloudLab preserves the dataset (it remains in the user's CloudLab account), and if a new experiment using the same profile is created, it can attach the same dataset with all previously written data. This is extremely useful for scenarios where a user wants to keep the data between runs. For instance, one could preload the dataset with VM images or datasets to be used in the OpenStack cluster; each new instantiation of the experiment can immediately access those without needing to download or copy them again. CloudLab ensures that local disk data can be persisted and loaded on future experiment nodes as long as the same profile/dataset is used.

From the user's perspective, the cloudlab_openstack Terraform resource behaves like any other Terraform resource: once applied, it will produce outputs so that the user can log into the OpenStack or run further automation (like Terraform OpenStack provider to create VMs inside that OpenStack). This represents a powerful abstraction – with one Terraform configuration, a user can go from zero to a running multi-node OpenStack testbed, including persistent shared storage, all through infrastructure-as-code.

Attribute	Required	Description
name	Yes	The name of your OpenStack experiment in CloudLab.
release	Yes	The OpenStack release version to deploy. Available options are: zed ubuntu22 (latest in this profile) xena wallaby
compute_node_count	Yes	Number of compute nodes to allocate for the OpenStack deployment.
os_node_type	Yes	The CloudLab hardware type for the nodes (e.g., m510, c220g1, xl170).
ml2plugin	Yes	The ML2 networking plugin to use. Use openvswitch

Figure 3.9: cloudalb_openstack_experiment Attributes

3.3 Simpler example of creating multiple similar experiments

In many research workflows, you often need to spin up a collection of CloudLab experiments that share the same baseline configuration-differing only in name or minor parameters. Manually duplicating resource blocks for each experiment quickly becomes tedious and error prone. By leveraging Terraform's iteration features (such as 'for_each'), you can define a single resource template that dynamically generates any number of experiments. Below is an example showing how to configure three identical experiments with one concise block; simply tweak the 'experiment_count' or adjust the per-instance attributes in the 'locals' map to scale up or customize your setup.

```
terraform {
  required_providers {
    cloudlab = {
      source = "pgrigo01/cloudlab"
      version = "1.0.0"
    }}}
  provider "cloudlab" {
    project         = "UCY-CS499-DC"
    credentials_path = "cloudlab-decrypted.pem"
    browser= "chrome"
  }
locals {
  experiment_count = 3
}
# Generate experiments 1 through experiment_count
# You can adjust `aggregate`, `image`, `node_count`, etc. in the `for_each`
    map below if needed.
resource "cloudlab_simple_experiment" "experiments" {
  for_each   = { for n in range(1, local.experiment_count + 1) :
      "experiment${n}" => n }
  name       = each.key
  routable_ip = true
  image      = "UBUNTU 24.04"
  aggregate  = "emulab.net"
  node_count = 1
}
```

Listing 3.3: Example Terraform configuration

4 Design and Implementation

4.1 System Overview

In this chapter, we present the architecture of the custom CloudLab Terraform provider system. We describe the system components, their interactions, and the overall flow of control when a user invokes Terraform to manage CloudLab resources. The architecture integrates the concepts discussed in the previous chapter: Terraform's plugin (provider) mechanism, a Flask-based backend service, and Selenium-driven automation for the CloudLab portal. Figure 4.1 illustrates the high-level architecture, showing how Terraform, the provider plugin, the Flask service, and CloudLab's interfaces work together.

The Terraform CLI/Core communicates with the custom CloudLab provider (a Go plugin), which in turn interacts with the CloudLab portal through a Python Flask API service. The Flask service invokes CloudLab's Portal API directly when possible, or uses Selenium to automate the CloudLab web interface for operations not exposed via the API. CloudLab's backend then provisions or tears down the experiment resources accordingly, and status or results are returned back up the chain.

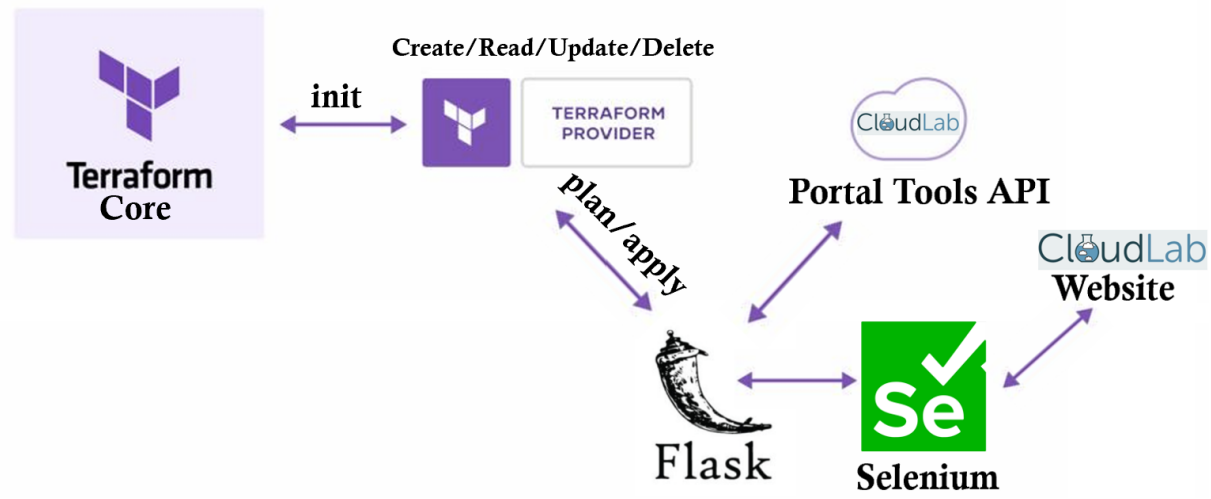

Figure 4.1: System Overview

4.2 Provider and Flask Request Workflow

Rather than embedding CloudLab API logic directly in Go, our provider serializes each create/delete call into an HTTP/JSON request and sends it to a local Flask service that will start automatically when the first terraform apply command is executed. When Flask receives such a request it dispatches to the correct handler and issues the corresponding CloudLab Portal API call. For creation, Flask issues a start experiment request, then polls CloudLab's status endpoint every few seconds until the experiment reports Active (or an error). Then it returns the experiment's ID and any other metadata back to the provider, which updates Terraform's state so that Terraform only proceeds once provisioning actually succeeded.

When terraform destroy runs, the provider sends an HTTP DELETE to /experiments/id on Flask. Flask calls CloudLab's terminate API, polls until all experiments are fully torn down, and returns success or error. The provider then removes the resources from 'terraform.tfstate'. Because Terraform may operate on many resources in parallel, both the provider and the Flask service are designed for safe concurrent requests.

Splitting responsibilities in this way keeps our provider plugin lightweight; it simply bridges Terraform and our execution logic, while the Flask service handles all CloudLab-specific interactions. That separation makes development and debugging much simpler (you can run and test Flask independently of Terraform) and lets us enhance API calls, retry logic, or automation strategies by updating only the Flask code. Architecturally, Terraform plus the provider form the control plane (making high-level decisions), and Flask acts as the execution plane (driving CloudLab). All communication happens over HTTP/JSON on localhost for maximum simplicity, and performance is never a bottleneck compared to CloudLab's own provisioning time.

4.3 Flask API Service for CloudLab Integration

One of the key architectural components of the system is a custom Flask-based web service that bridges Terraform to the CloudLab platform. CloudLab does not have a complete public REST API, thus we created this Flask API tool to serve as a middleman between Terraform provider (written in Go) and CloudLab's Portal Tools API to script experiment interactions [12] using a lightweight Python HTTP layer [14]. The Flask service runs locally (on localhost:8080) and exposes HTTP endpoints for experiment operations (creation, status query, and termination).By means of these endpoints, the Terraform provider's client code (Section 3.2.2) passes the complexity of authentication and web interaction to the Flask layer. In this section, we describe the design of the Flask API service, its endpoints, and its logic.

4.3.1 Design And Role

The Flask service is a lightweight web server (implemented in Python using the Flask micro framework) that the provider launches on-demand. [14] When a Terraform operation (apply or destroy) requires CloudLab interaction, the provider ensures the Flask service is running (using the ensureFlaskRunning() method) and then issues an HTTP request to the appropriate endpoint. By decoupling CloudLab-specific logic into this service, we keep the Go provider code simple, the provider merely sends requests and handles responses, while the Flask service performs the actual experiment orchestration (including any needed browser automation). This separation also makes maintenance easier, as CloudLab-specific changes.

4.3.2 Why Implement Flask as a Server Instead of Script Invocation

At first glance it might seem simpler for our Go provider to execute:

```
python myscript.py
```

on each API call, but this approach runs into several practical limitations when integrated with Terraform's plugin architecture:

1. **Avoidance of per-call startup costs.** Spawning a new Python process on every request incurs heavy overhead-creating a shell, launching the interpreter, parsing the script and its dependencies. A long-running Flask server pays that cost only once at startup and then serves each request almost immediately.
2. **Connection setup and certificate caching.** Each CloudLab XML-RPC call requires the loading and parsing of the user's X.509 certificate, the negotiation of a mutual TLS handshake (including the handling of CloudLab's non-standard AIA extension), and the construction of a new HTTP client. Forking a Python process per call repeats this full crypto and handshake cost, slowing down large Terraform plans and preventing any credential caching. A persistent Flask process can load the certificate once, maintain an authenticated session, and reuse its HTTP/XML-RPC client across dozens of Create/Read/Delete calls.
3. **Managing long-running workflows and retries.** Real-world CloudLab operations often block for tens of seconds (polling until an experiment is *Active*, retrying transient timeouts, driving Selenium browser flows). Embedding those waits inside Terraform's Create/Delete handlers risks gRPC timeouts or deadlocks. Offloading blocking work to Flask lets our Go plugin issue a single 'startExperiment' HTTP request and either poll lightly or return immediately, while Flask handles sleep loops, retries, and browser automation behind the scenes.
4. **Stateful coordination across resources.** Terraform plans can create, update, and destroy dozens of experiments in parallel, and workflows like our extension algorithm need global state (e.g., "what is the latest expiry time across all active experiments?"). A single in-memory Python process naturally holds shared state, via APScheduler jobs, CSV caches, or internal maps, whereas isolated script invocations would scatter state on disk or require complex inter-process communication.
5. **Cleaner separation of concerns.** Terraform's Go plugin SDK excels at schema validation, state management, and resource graph orchestration, but lacks mature XML-RPC tooling for CloudLab's quirks (AIA extensions, legacy SHA-1 certs) and browser automation capabilities. By treating Flask as the "execution engine" (CloudLab logic, Selenium scrapers, retry loops) and the Go provider as the 'control plane' (Terraform schema, de-

pendency graphs, state reads/writes), each component stays small, focused, and easier to maintain.

In short, running a dedicated Flask service provides persistent connections and certificate caching, efficient handling of long-running tasks and retries, a natural home for shared, in-memory state, and a clear modular separation, without forcing Terraform Core or our Go plugin to repeatedly spawn heavyweight Python logic.

4.3.3 Authentication via User Certificate

An important job the Flask has to do is to handle CloudLab authentication securely. CloudLab uses user X.509 certificates for programmatic access (as described in CloudLab's 'Portal API' documentation). To do this, the user supplies their CloudLab certificate (PEM file) path in the Terraform provider configuration (credentials_path). The provider passes this certificate file to the Flask service on each request . The Flask service then uses the certificate to authenticate with CloudLab's backend Portal API. In practice, the Flask code loads the PEM certificate to establish a client-authenticated connection to CloudLab. This allows the service to invoke any available CloudLab API endpoints (via an XML-RPC call using Python's built-in xmlrpc.client module [15]) on behalf of the user.

4.3.4 Flask Integration: Design Decisions

We opted for Flask to leverage CloudLab's native Python automation SDK and Portal API directly avoiding any need to reimplement or wrap their tooling in another language. Because the entire Portal Tools suite is Python-based, building our integration layer in Flask gave us seamless access to its functions. Building a Go HTTP server instead would have forced us to either reimplement CloudLab's API in Go or shell out to Python scripts-both of which add complexity and invite subtle bugs.

In early attempts to eliminate Flask, we experimented with several Go libraries. We tried the github.com/kolo/xmlrpc client and alternative parsers such as ZMap's zcrypto/x509 package [ZCrypto], hoping they would accept CloudLab's unusual Authority Information Access (AIA) extension (OID 2.25), which tells clients where to fetch issuer certificates. In every case, however, Go's strict x509 parser refused to parse the AIA entry and couldn't establish an authenticated XML-RPC channel, since the server's legacy SHA-1 signatures are now considered insecure.

We then extracted the certificate and private key from CloudLab's PEM bundle and generated a new self-signed cert with SHA-256. Although this gave us a stronger signature (sha256), it remained self-signed and therefore failed to connect to Emulab's real CA. Pointing Go's TLS

client at our modified bundle still broke the handshake, because the server continued to present its official certificate.

In short, CloudLab's combination of an unrecognized AIA extension plus a legacy SHA-1 signature is something Go simply refuses to accept. Python's OpenSSL bindings, by contrast, handle these certificates without complaint. By embedding our Terraform provider calls in a lightweight Flask service, we offload all of the certificate handling, XML-RPC invocation, and authentication to well-tested Python libraries—avoiding duplication of complex logic and ensuring rock-solid integration with CloudLab's standard workflow.

4.3.5 Endpoints

To bridge Terraform with CloudLab's Portal API, our Flask service exposes three RESTful endpoints `POST /experiment`, `GET /experiment`, and `DELETE /experiment` which correspond to starting a new experiment, polling its status, and terminating it, respectively. Each endpoint accepts the user's uploaded certificate and Terraform parameters, validates the required fields, and configures an authenticated XML-RPC client against CloudLab's control interface. It then invokes the appropriate Portal API call, employs retry logic to mitigate intermittent timeouts, and returns a clear HTTP response (or JSON payload) that Terraform uses to drive provisioning decisions. This structure cleanly encapsulates all CloudLab-specific integration logic in Python, leaving the Terraform provider focused solely on state management and workflow orchestration.

POST /experiment - startExperiment

- **Extracts inputs:** Reads the user's uploaded certificate and the Terraform parameters.
- **Validates required fields:** Ensures both the CloudLab project and profile identifiers are present; returns an error if either is missing.
- **Configures the CloudLab client:** Establishes a secure, authenticated connection to CloudLab's control interface using the certificate.
- **Launches the experiment:** Sends the 'start experiment' request to CloudLab, retrying a few times to handle issues with timeout that the Portal API has sometimes before giving a valid answer.
- **Captures the experiment ID:** Retrieves the new experiment's unique identifier from the response or via a quick status call if it's not immediately available.
- **Returns success or failure:** Sends back a simple success message (including the experiment's UUID) or an error, which Terraform uses to determine provisioning outcome.

GET /experiment - experimentStatus

- **Extracts inputs:** Reads the user's uploaded certificate and the Terraform parameters.
- **Validates required fields:** Ensures both the CloudLab project and profile identifiers are present; returns an error if either is missing.
- **Configures the CloudLab client:** Establishes an EmulabXMLRPC connection using the provided certificate.
- **Retrieves status** Calls api.experimentStatus until a valid response is returned.
- **Returns status or error:** On success, returns the raw 'response.output' string and the HTTP status mapped from the exit code.

DELETE /experiment - terminateExperiment

- **Extracts inputs:** Reads the user's uploaded certificate and Terraform parameters.
- **Normalizes experiment identifier:** Ensures that the request's 'uuid' or the 'project,experiment' combination is formatted correctly for the Portal API.
- **Validates required fields:** Confirms that both 'proj' and either 'uuid' or 'experiment' are present; returns 400 if missing.
- **Configures the CloudLab client:** Establishes an 'EmulabXMLRPC' connection using the provided certificate, with SSL verification disabled.
- **Terminates the experiment by calling api.terminateExperiment.**
- **Returns success or failure:** On first zero exit code, returns OK with HTTP 200, if all retries fail, returns the mapped error message and its HTTP status code.

4.4 Terraform Provider Plugin Design (Go)

The CloudLab Terraform provider is implemented in Go with the Terraform Plugin SDK (v2) from HashiCorp [16]. As a standard Terraform provider plugin, it encapsulates CloudLab's data model-projects, profiles, experiments, networks and exposes each as a first class Terraform resource. This lets Terraform handle lifecycle (create, read, update, delete) and state management automatically, just as it does for AWS, GCP, or Azure resources. Once published to the Terraform Registry, users install it via terraform init and can reference resources like cloudlab_project in their plans.We can see the provider publication below:

Under the hood, the provider implements the SDK's core -interfacesProvider, Resource, and related ConfigureContextFunc/ValidateConfigFunc callbacks to manage configuration, CRUD operations, and state file interactions. By conforming to these well defined interfaces, the code remains modular and reusable: any future CloudLab resource simply needs its own Resource implementation, and it slots straight into the provider without rewriting the plumbing.

The CloudLab provider must authenticate to CloudLab on the user's behalf using a PEM-encoded client certificate. When users sign up for CloudLab, they download a .pem file that serves as their identity for API calls. To streamline setup, we provide an install script that installs Go, Terraform, and any Python/Flask dependencies, downloads the user's encrypted credentials, decrypts them, and places the resulting .pem in your workspace. Our Terraform schema then exposes this via a credentials_path attribute, so once your .pem is in place the provider knows exactly where to find it.

Under the hood, when you run terraform apply, the provider will automatically launch its Flask-based helper server if it isn't already running, wait for that service to start, and then send all API requests ,passing the credentials_path so the service can load the PEM and authenticate to CloudLab. In addition to credentials_path, the schema also includes options for the CloudLab project ID (project) and the headless browser (browser, either Chrome or Firefox) used to fetch global state.

This is how our provider is defined as shown below:

```
terraform {
  required_providers {
    cloudlab = {
      source = "pgrigo01/cloudlab"
      version = "1.0.0"
    }
  }
}
provider "cloudlab" {
  project        = "UCY-CS499-DC"
  credentials_path = "cloudlab-decrypted.pem"
  browser        = "chrome"
}
```

Listing 4.1: Example Terraform configuration

Under the hood, the provider plugin implementation follows Terraform's standard CRUD pattern for resources:

Create: If a user defines a `cloudlab_simple_experiment` resource, the provider's Create function sends a `POST /experiment` request to start a new experiment to the flask server.Then it passes the user's PEM certificate (via the `credentials_path`) and the parameters from the Terraform schema. Our Flask endpoint then parses the XML-RPC response to Cloudlabs' Portal API, and returns either the newly assigned experiment ID or an error.

Read: The provider defines read (or refresh) logic to query CloudLab for the current state of an experiment resource. The provider sends a `GET /experiment` call to fetch the experiment's status, so that Terraform can update its state file and remain aware of any out-of-band changes.

Update: True in-place updates in the CloudLab Terraform provider are not currently feasible because CloudLab's portal-tools and public API only support starting, querying, and terminating experiments - there is no updateExperiment endpoint to call. Terraform's in-place update lifecycle (the Update RPC) requires a corresponding remote API call, so without it the provider must treat any changed argument as requiring full resource replacement. In practice, this means using Terraform's lifecycle meta-arguments (e.g., create_before_destroy) or to tear down and recreate the experiment rather than update it in place.

Although CloudLab offers an experimental `modifyExperiment` API, we chose not to invoke it: under the hood it still tears down and reconstructs most resources (so it is not a true live reconfiguration), and it is not part of the stable, supported API, making its behavior unpredictable.

Instead, our OpenStack-based profile achieves elasticity by setting `client.elastic = true` in the provider's `Configure` phase. Each new `cloudlab_openstack_experiment` resource then:

- Reuses the same pre-provisioned 100 GB persistent dataset, and
- Joins the same VLAN on both control and compute networks.

Users scale out simply by adding more `cloudlab_openstack_experiment` blocks-each launches its own experiment, yet all share storage and networking seamlessly, without any in-place modifications to running experiments.

As a result, true elasticity is realized by provisioning additional experiments on demand rather than mutating an existing one, preserving the familiar Terraform workflow: edit your HCL file, run `terraform apply`, and the cluster grows automatically.

Delete: The provider's Delete implementation triggers experiment termination on CloudLab (it sends a `DELETE /experiment` request). This corresponds to the user terminating an experiment, which the provider accomplishes via the CloudLab API.

4.4.1 Provider Implementation (provider.go)

The provider.go module defines the Terraform provider itself, the Go-based plugin that bridges Terraform's core engine with CloudLab. In its Provider() function and accompanying ConfigureContextFunc, it declares the provider schema (project ID, PEM path, chosen browser) and validates that each required field or environment variable is present. Once configured, it starts a client that communicates with our Flask for all CloudLab interactions.

During terraform apply, Terraform invokes each resource's CRUD callbacks as defined by the Plugin SDK. The provider marshals HCL configuration into JSON requests, forwards them to the Flask helper for execution (handling authentication, XML-RPC calls, retry loops, and any browser automation), and then integrates the JSON responses back into Terraform's state. Because all platform-specific logic lives in the Flask layer,provider.go remains concise and focused on high-level orchestration. New CloudLab resource types can be added simply by implementing a Go resource.Resource that plugs into this provider—no changes to the core plumbing are required.

New() constructor: Returns the plugin version. When Terraform initializes the plugin, it calls this function to create a provider instance. Capturing version here ensures consistent version reporting across diagnostic output and registry queries.

```
func New(version string) func() provider.Provider {
    return func() provider.Provider {
        return &cloudlabProvider{
            version: version,
        }}}
```

Listing 4.2: Provider constructor in Go

Schema definition: We declare three top-level attributes:

- `credentials_path` (string, required, sensitive): file path to the user's PEM certificate for API auth. Marking it sensitive prevents it from leaking in logs.
- `project` (string, required, sensitive): the CloudLab project UUID under which experiments run.
- `browser` (string, required with validator): selects the automated browser to launch (Chrome or Firefox).

```
func (p *cloudlabProvider) Schema(_ context.Context, _ provider.SchemaRequest,
    resp *provider.SchemaResponse) {
    resp.Schema = schema.Schema{
        Attributes: map[string]schema.Attribute{
            "credentials_path": schema.StringAttribute{
                Required: true,
                Sensitive: true,
            },
            "project": schema.StringAttribute{
                Required: true,
                Sensitive: true,
            },
            "browser": schema.StringAttribute{
                Required: true,
                Sensitive: false,
            },},}}
```

Listing 4.3: Provider schema definition in Go

Configure() method:

First, we read the user's configuration into a Go struct. Then we validate that required attributes were not left 'unknown' .In our case on our provider block we must define the credentials_path, the browser and the project under which our experiments are running. After validating that none of these attributes is missing we build and expose our CloudLab Client. Finally we tell terraform what resources we offer.

4.4.2 Client Implementation (client.go)

client.go handles all communication between the Terraform provider and CloudLab.It starts the flask server (using either Chrome or Firefox for automation), it forms requests with the appropriate authentication, and parses answers. . It implements core operations like starting experiments, checking their status, and terminating them.

We start by defining the Client struct. This struct holds two mutable runtime flags:

elastic (bool): When true, the client uses CloudLab's default experiment profile. When false, it switches to the OpenStack profile.

serverType : Specifies which Flask endpoint the client will communicate with.

func (c *Client) ensureFlaskRunning() Verifies the Flask helper is listening on port 8080; if not, when terraform apply is run it starts the appropriate Python server script and waits for it to become available.

func mapToJSON(data interface) () Transforms a Go value (map or struct) into a JSON-formatted string, returning an error on failure.

func (c *Client) sendRequest() This function ensures Flask is running, constructs a multipart form with top-level fields and a JSON "bindings" blob, attaches the credentials file, executes the request, and returns the response body, status code, and any error.

func (c *Client) startExperiment() Wrapper that calls 'sendRequest' with POST to create a new experiment and returns the raw response or an error.

func (c *Client) terminateExperiment() Wrapper that calls 'sendRequest' with DELETE to terminate an experiment by its identifier.

func (c *Client) experimentStatus() Wrapper that calls 'sendRequest' with GET to retrieve experiment status, parses the plaintext response into a key/value map, and maps it to internal status codes.

4.5 Selenium Automation of CloudLab Portal

As described in Section 2.5, we leverage Selenium in two specialised ways to bridge gaps in CloudLab's Portal API:

We keep these browser-driven tasks strictly isolated in the Flask layer so that all other provisioning and lifecycle calls use the more robust CloudLab Portal Tools API. This hybrid approach ensures that these missing functionalities from the official API are still handled programmatically, without forcing UI-driving into everyday experiment management.

1. **Automated Credential Management.** Using the user's portal username and password, provided once via an environment variable or prompt, a headless browser session logs into the CloudLab portal on provider initialization navigates to the account settings page and downloads the X.509 certificate file (cloudlab.pem). The same script calls OpenSSL to decode this file into cloudlab-decrypted.pem without human intervention, which our Flask intermediary then uses for all later XML-RPC requests. The load and risk of hand certificate handling is eliminated by this totally automated credential system.
2. **Global State Reading.** Periodically loads the dashboard 'My Experiments' in the headless browser, sorts the table of active experiments (fields including Name, Profile, Status, Created, and Expires), and writes the consolidated data into a local CSV.

We keep a clear separation between browser-driven chores and the main CloudLab automation logic by restricting Selenium to these two roles-credential management and global state scraping, and assigning all basic provisioning calls to our Flask API. Without depending on brittle UI-driving for daily experiment lifecycle operations, this hybrid approach guarantees that every required portal interaction is handled programmatically.

4.6 Experiment Extension Algorithm

To ensure that all active CloudLab experiments remain online for the duration of the longestr-running experiment in a given project, we developed an *Experiment Extension Algorithm*. This algorithm aperiodically checks the expiration times of all experiments, computes the latest expiration timestamp, and automatically extends any experiment whose lifetime would otherwise expire significantly earlier than that maximum. The following subsections describe its components and workflow.

4.6.1 Overview

The goal of the extension algorithm is to reduce manual intervention in maintaining experiment lifetimes. By aligning all experiments' expiration times to the latest one, users can avoid unexpected terminations of experiments launched at different times. The algorithm operates in four main phases:

1. **Data Collection:** Refresh the list of running experiments.
2. **Expiration Update:** Fetch and parse up-to-date expiration timestamps.
3. **Decision Logic:** Identify experiments needing extension based on a configurable threshold.
4. **Extension Execution:** Invoke the CloudLab extension command with retry logic.

4.6.2 Data Collection and Expiration Update

In the first phase, we invoke the `chromeExperimentCollector.getExperiments` (or its Firefox equivalent) to scrape the current list of experiments via the CloudLab portal. This updates a local CSV file (`cloudlab_experiments.csv`) with each experiment's metadata (project name, experiment name, etc.).

Next, `getCSVExperimentInfo.getCSVExperimentsExpireTimes` reads that CSV and produces `experiment_expire_times.csv`, which includes a column of human-readable expiration timestamps obtained from the portal API.

4.6.3 Parsing Timestamps

Each row of `experiment_expire_times.csv` contains a timestamp string. We use the helper function

```
parse_expire_time(expire_str)
```

which parses:

- `datetime.strptime(expire_str, "%Y-%m-%d %H:%M:%S")`

All resulting `datetime` objects are normalized to UTC to allow correct comparison across time zones.

4.6.4 Decision Logic

After loading the list of experiments and their parsed expiration times into memory, the algorithm:

1. Computes
$$T_{\text{max}} = \max_{e}\{\text{expire\_time}(e)\}$$
where the maximum is taken over all experiments e.
2. For each experiment e with expiration time $T_e < T_{\text{max}}$, calculates the difference in hours
$$\Delta_h = \frac{T_{\text{max}} - T_e}{1 \text{ hour}}.$$
3. Compares Δ_h against a user-configurable threshold h_{thresh} (default: 1.0 h). If $\Delta_h < h_{\text{thresh}}$, the algorithm skips extension to avoid trivial adjustments.
4. Otherwise, computes the extension amount
$$H = \lceil \Delta_h \rceil$$
(rounding up to the next whole hour) and proceeds to the extension phase.

4.6.5 Extension Execution

For each selected experiment, the algorithm calls:

```
extendExperiment.extend_experiment(
  project_and_name = "Project,ExperimentName",
  hours_to_extend   = H,
  message           = "Extending to match latest expiration"
)
```

This function wraps the CloudLab `extendExperiment` CLI command and implements:

- **Retry Logic:** Up to `MAX_RETRIES` (default 5) attempts with a `RETRY_DELAY` pause, to handle transient SSL or network errors.
- **Empty-Response Handling:** Treats an empty stdout response as a successful extension.
- **Error Reporting:** Logs non-retryable errors and aborts further attempts for that experiment if necessary.

4.6.6 Integration and Scheduling

The entire algorithm is exposed via the Flask intermediary API and invoked automatically by an `APScheduler` background job every hour. This scheduler also refreshes experiment data on the same interval, ensuring that any new experiments are enrolled in the extension process without manual triggers. Together, these components provide a robust, hands-off solution for keeping all CloudLab experiments within a project running synchronously until the last one terminates.

By implementing this algorithm, we achieve continuous alignment of experiment lifetimes, minimize the risk of premature terminations, and reduce the operational burden on researchers conducting long-running multi-experiment studies.

4.7 Summary

To conclude this chapter, our thesis involved integrating several components to realize the custom CloudLab Terraform provider: Our approach started with a careful study of CloudLab's current systems. We looked at the manual portal, the XML-RPC API (portal tools), available GENI client libraries, and portal-tools to determine which operations could be automated directly. This enabled us to pinpoint the main chores, experiment development, status monitoring, renewal, and teardown, that could use the API as well as the edge-case interactions calling for browser automation.

We then created a Terraform provider skeleton with Terraform's Go SDK. We described CloudLab experiments and networks as resource types within the provider plugin, so capturing all high-level state management and CRUD (Create, Read, Update, Delete) operations. Keeping this logic in Go helped us to guarantee flawless integration with Terraform's planning and apply processes, so laying the foundation for offloading more difficult chores to outside assistants.

To handle certificate-based XML-RPC calls and other multipart operations where Python tooling is more mature, we developed a minimal Flask web service to run alongside the Terraform provider. This helper service exposes JSON-driven HTTP endpoints-such as /create-experiment and /renew-that accept the necessary parameters and return the operation status or identifiers. We use standard requests and XMLrpc modules in addition to already existing Python CloudLab/GENI client libraries within Flask to avoid having Go reimplementing authentication, disk-image management, or polling logic. Most importantly, this separate process handles long-running tasks such as waiting for an experiment to be ready, maintaining the Terraform thread responsive, and preventing provider timeouts.

For the remaining interactions not covered by any API, specifically credential acquisition and global state inspection, we integrated Selenium WebDriver into the Flask service. In a headless browser session, Selenium logs into the CloudLab portal using environment-supplied credentials, navigates to the account settings page, and programmatically downloads and decrypts the X.509 certificate needed for XML-RPC authentication.Separately, it writes these data to a local CSV after reading the "My Experiments" dashboard at regular intervals and extracting experiment metadata (including name, profile, status, creation, and expiration timestamps).

We produced a modular, maintainable design by precisely separating concerns: letting the Terraform provider manage state, the Flask service run direct CloudLab operations in Python, and Selenium handle some browser operations. This system allows us to keep the Go codebase lean, prototype quickly, and independently improve every component. In the next chapter we will show this three-tier architecture and follow a complete Terraform apply, demonstrating how Terraform, Flask, and Selenium cooperate to control a CloudLab experiment lifecycle.

5 Evaluation

5.1 Cloudlab Web vs Terraform Provider

In this section, we present an evaluation of our Terraform-based CloudLab provider against the native CloudLab Web interface. We show the trade-offs that accompany each method by looking at how a single Terraform command simplifies deployment and teardown relative to manual portal interactions, evaluating the learning curve and user guidance given by HCL against the CloudLab GUI, and measuring consistency and responsiveness under both workflows. Our goal is to show not only where Terraform offers clear advantages, but also where the CloudLab Web interface remains more intuitive or flexible in some cases.

From the comparisons below , it is clear that when automation, repeatability, and perfornance are top concerns, Terraform for CloudLab excels. It drastically reduces repetitive tasks and possible discrepancies in the execution of tests at the expense of requiring users to implement an Infrastructure as Code (IaC) approach and set up the necessary tools. The CloudLab portal remains a handy choice for one-time, brief uses or for using features not yet accessible via the provider of Terraform. Actually, we see power users using Terraform for the bulk of their workflow, particularly with regard to iterative experiments and resource coordination among several sites, perhaps sometimes using the CloudLab GUI for particular chores (such as debugging or profile production) as necessary.

Table 5.1: Comparison of Terraform and CloudLab Web

Aspect	Terraform		CloudLab Web	
	Pros	**Cons**	**Pros**	**Cons**
Automation	One-command deployment and teardown.	Requires setup of Terraform and environment.	No setup needed beyond a web browser.	Manual clicks for each action; cannot easily automate sequence.
Ease of Use	Once written, HCL + plan/apply guides you and avoids mistakes.	Steeper learning curve for Terraform syntax and state.	Intuitive GUI for those unfamiliar with IaC tools; immediate visual feedback (progress bars, etc.).	Repetitive for frequent tasks, prone to misclicks (e.g., clicking wrong profile).
Repeatability	Code versioning and shareable configs yield identical reprovisions.	State drift if configs/state not managed carefully.	Profiles ensure consistent environments.	Each instantiation is manual; no bulk-run support.
Performance	Faster when creating multiple experiments.	Limited live feedback; must wait for final CLI prompt.	Portal interactions are responsive; user perceives progress in real time.	Users often idle waiting for provision.
Scalability	Code can define many experiments or complex topologies in one config.	Very large HCL files can be hard to maintain.	Supports more profiles/topologies to create experiments.	Scaling multiple experiments remains manual and time-consuming.

5.2 Experimental Results

To evaluate the performance of the Terraform-based CloudLab provider against the traditional CloudLab web interface, a series of benchmarks were performed that involved multiple experiment deployments. The goal was to assess the system's behavior both when provisioning experiments sequentially and in batch mode, simulating realistic user workflows.

5.2.1 Parallel Experiment Provisioning

When Terraform provisions experiments in parallel, it issues multiple create requests concurrently. This approach significantly reduces the overall time required to bring a batch of experiments online, as the operations proceed independently after an initial setup phase. In contrast, the CloudLab web interface requires users to submit experiments one at a time. Each submission must be completed and become fully operational before the next can be started. As a result, the total time required to create a series of experiments using the UI increases proportionally with the number of experiments.

The Terraform provider's ability to manage multiple experiments concurrently results in a noticeable improvement in provisioning efficiency. However, the time at which each individual experiment completes can vary depending on site load, image availability, and other backend conditions. This variance is a trade-off for the gain in aggregate throughput.

5.2.2 Sequential Per Experiment Provisioning

To compare the provisioning behavior of one-to-one experiments, both methods were tested under sequential conditions. In this mode, Terraform was configured to handle one experiment at a time, replicating the same workflow as the UI. Under these circumstances, both provisioning methods exhibited comparable completion times. Terraform consistently completed experiments with minimal delay and demonstrated similar or slightly better stability between runs compared to the UI. This suggests that for users deploying a single experiment, such as for debugging or small-scale testing, both Terraform and the CloudLab interface offer reliable provisioning performance. Terraform, however, retains the advantage of automation, repeatability, and integration into Infrastructure As Code workflows.

To further quantify this performance difference, time-based benchmarks were conducted comparing the two approaches, from single experiment deployments up to 40 concurrent instances. The results show that Terraform consistently outperforms the CloudLab web interface, especially in batch provisioning scenarios. For example, provisioning 40 experiments using Terraform took less than 3 minutes, while the CloudLab website, even with a best-case estimate of 30 seconds per experiment, required around 20 minutes. This highlights Terraform's significant time efficiency and scalability benefits, making it the preferred tool for large-scale or automated experimental workflows.

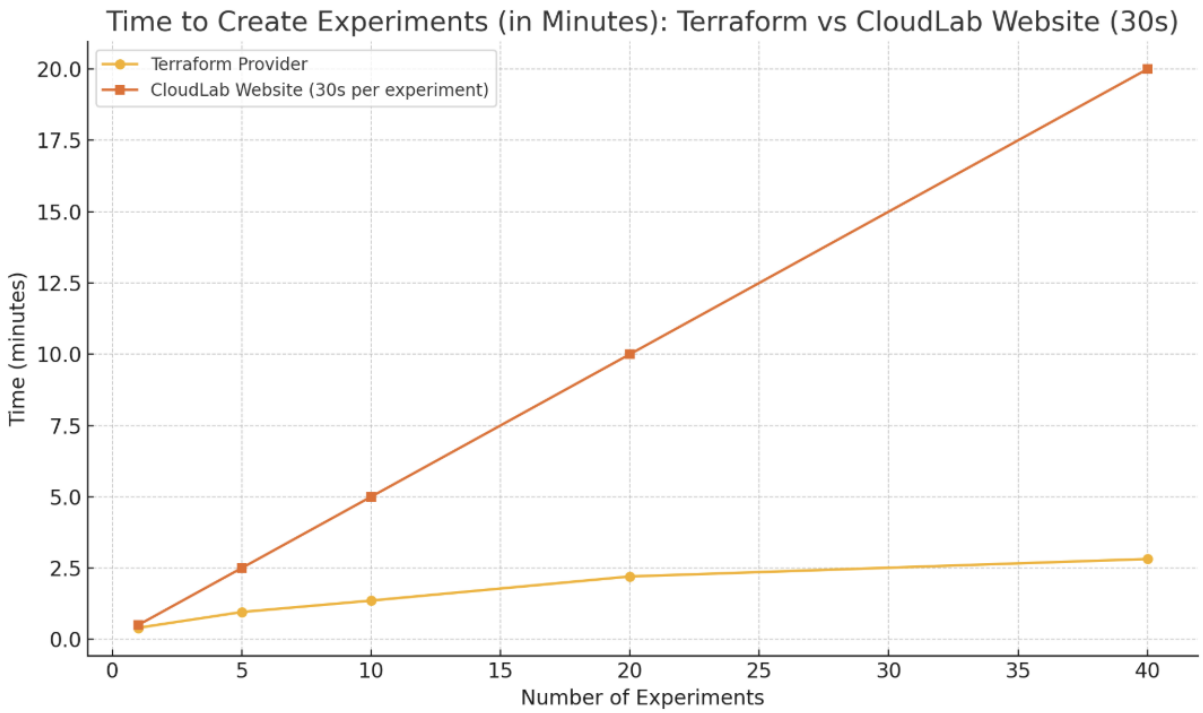

Figure 5.1: Time Comparison

5.3 Discussion

Provisioning Strategy and Performance. Terraform's parallel provisioning model offers substantial performance benefits when launching multiple experiments. This approach minimizes user waiting time and improves system responsiveness, especially for workflows involving multiple concurrent deployments. However, when provisioning is performed sequentially, both the UI and Terraform exhibit similar response times, indicating parity in single-instance use cases.

Consistency and Variability. While parallel provisioning with Terraform introduces some variability in individual experiment completion times, this behavior is expected due to asynchronous resource allocation and differing backend site conditions. Despite this, the total provisioning duration remains significantly shorter than the sequential alternatives.

Usability and Automation. One of the key benefits of Terraform lies in its automation and reproducibility. Users can define the infrastructure declaratively and version their configurations, which is not possible through the manual UI process. This makes Terraform more suitable for large-scale, repeatable experiments and collaborative environments.

6 Limitations

6.1 CloudLab Platform Limitations

Experiment Elasticity and Modification: CloudLab's approach to experiments emphasizes fixed, repeatable environments over dynamic elasticity. Once an experiment is instantiated, it is not trivial to change its resource allocation without terminating and re-instantiating. This contrasts with public clouds where one can often resize or scale resources on the fly. Our Terraform provider inherits this limitation. For instance, if a user wishes to scale up a CloudLab experiment by adding nodes via Terraform, the provider cannot simply call an API to add a node; instead, the user must modify the Terraform configuration (e.g., increase a count) which will cause Terraform to delete and recreate the experiment. This leads to experiment downtime and is less seamless than scaling a typical cloud deployment. In Terraform, we mark such changes as requiring replacement rather than in-place update. This behavior is aligned with CloudLab's design (complete environment descriptions), but this means that our provider does not support incremental changes to a running experiment. It is a fundamental limitation rooted in CloudLab's less elastic nature.

Authentication and Security Constraints: CloudLab's authentication via user certificates is powerful but also cumbersome. The need to handle a private key and certificate means that our provider requires careful Authentication and Security Constraints: CloudLab authenticates API requests using the X.509 user certificate.

While this is secure, it complicates the usage of our provider. The user must securely provide their certificate and key, and our system must handle them. Unlike cloud providers where Terraform can use short-lived API tokens or integrate with cloud SSO, CloudLab's certificate is a long-lived credential that needs careful handling. We mitigated this by not embedding the certificate in Terraform state (we only store its file path), but the approach still requires the certificate file to be present on disk. This could be a limitation in environments like Terraform Cloud or remote execution, where providing a file securely is non-trivial. Furthermore, any compromise of the machine running the Terraform provider (or the Flask service) could potentially expose the user's CloudLab credentials. We have not implemented additional security layers (such as prompting for a password to decrypt the certificate); doing so would break non-interactive au-

tomation. Thus, the onus is on the user to protect the certificate file. This limitation inherits CloudLab's authentication mechanism and contrasts with many Terraform providers that use revocable API keys.

6.2 Go Limitations

Terraform providers are compiled Go binaries that Terraform Core loads to manage the infrastructure. However, CloudLab's control API is implemented in Python and communicates through XML-RPC-sending and receiving XML-formatted function calls over the network. Although Go offers some XML-RPC libraries, none support the features required by CloudLab's existing Python client. As a result, a direct Go-to-CloudLab connection would miss important capabilities, leading to errors or unsupported operations [17, 18].

To bridge this gap, an intermediary service is written in Python. When the Terraform provider (in Go) needs to perform an action on CloudLab, it sends a simple request- over HTTP to the Python bridge. This intermediary then translates the request into the precise XML-RPC calls that CloudLab's API expects, invoking the necessary Python client functions. Once CloudLab responds, the bridge converts the XML-RPC reply back into a straightforward format (such as JSON) and returns it to the Go provider.

By introducing this Python bridge, you get the best of both worlds: the performance, type safety, and concurrency features of Go for your Terraform provider, and the full, battle-tested capabilities of CloudLab's Python XML-RPC client. Terraform itself never sees the Python code; it only interacts with the Go binary, preserving a clean separation of concerns and a standard Terraform workflow.

In the future, one might develop an XMLRPC library in Go that supports the certificate used by Cloudlab (which is using an older encryption algorithm) and is not considered safe by Go libraries. However, now the intermediary Python API offers a maintainable solution to ensure seamless integration between a Go-based Terraform provider and the Python-based CloudLab API.

7 Conclusions

7.1 Conclusion

In this thesis, we have presented the design and implementation of a fully featured Terraform provider for CloudLab, moving beyond the simple VM-creation prototype of previous work. By introducing two core resource types, cloudlab_simple_experiment for arbitrarily-sized multi-node clusters with optional blockstore volumes, and cloudlab_openstack_experiment for turnkey OpenStack deployments with persistent shared storage, we have enabled researchers to manage complex bare-metal and virtualized testbeds declaratively.

We use a lightweight Flask-based intermediary to handle authentication, XML-RPC interactions, and browser automation (via Selenium) for credential retrieval and global state scraping. This decoupling keeps the Go provider lean, while Python handles CloudLab's legacy certificate quirks and UI gaps. We have further automated lifecycle management with a background experiment-extension scheduler that aligns all active experiments' lifetimes, reducing manual intervention and preventing premature terminations.

Overall, this work demonstrates that CloudLab experiments can be first-class Terraform resources—fully automated, version-controlled, and composable with other cloud platforms. Researchers can now define, provision, and tear down entire experimental environments in a single Terraform workflow, achieving unprecedented reproducibility, scalability, and integration potential in cloud infrastructure research.

7.2 Future Work

- **Replacing the Flask server:** Replace the Python intermediary with a Go-native XML-RPC client that supports CloudLab's AIA extensions and legacy signatures, removing external dependencies and simplifying deployment. Right now Go lacks an official library that fully supports all the functionalities of Python's XMLRPC. Maybe in the future there will be better support for the legacy algorithm SHA1 the cloudlab.pem uses.
- **Expanded Profile Library:** Add more Terraform resources for additional CloudLab profiles (e.g., Kubernetes, Hadoop, GPU-accelerated nodes), enabling richer experimentation workflows directly from HCL.
- **In-place Update Functionality (conditional):** Design and implement an `updateExperiment` capability in the Terraform provider to allow modifying existing experiments without recreation. This will require CloudLabs' Portal Tool API [12] to expose a corresponding `updateExperiment` API endpoint-until then, most changes must continue to be handled via resource replacement.

BIBLIOGRAPHY

[1] Spacelift, "Terraform architecture," https://spacelift.io/blog/terraform-architecture, 2024, accessed: 2025-05-09.

[2] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K.-C. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. S. Kar, and P. Mishra, "The design and operation of CloudLab," in *Proc. USENIX Annual Technical Conference (ATC)*, 2019, pp. 1–14.

[3] CloudLab, "CloudLab user documentation," [Online]. Available: https://docs.cloudlab.us/, accessed: May 2025.

[4] G. Panis, "Managing cloudlab infrastructure with terraform," Bachelor's thesis, University of Cyprus, Nicosia, Cyprus, 2024, undergraduate thesis submitted to the Department of Computer Science, University of Cyprus.

[5] Johnson, D., "openstack-build-ubuntu," https://gitlab.flux.utah.edu/johnsond/openstack-build-ubuntu.git, 2025, gitLab repository, accessed 2025-05-13.

[6] Wikipedia contributors, "Infrastructure as Code," https://en.wikipedia.org/wiki/Infrastructure_as_code, 2025, [Online; accessed 18-May-2025].

[7] HashiCorp, "Terraform official website and documentation," [Online]. Available: https://www.terraform.io, accessed: May 2025.

[8] ——, "Terraform provider development documentation (sdk v2)," [Online]. Available: https://developer.hashicorp.com/terraform/plugin, 2023, accessed: May 2025.

[9] ——, "aws_instance resource," https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/instance, 2025, [Online; accessed 18-May-2025].

[10] Amazon Web Services, "Amazon Simple Storage Service (S3)," https://aws.amazon.com/s3/, 2025, [Online; accessed 18-May-2025].

[11] Selenium Project, "Selenium webdriver documentation," [Online]. Available: https://www.selenium.dev/documentation, accessed: May 2025.

[12] L. Stoller, "Portal tools for CloudLab," GitLab repository. [Online]. Available: https://gitlab.flux.utah.edu/stoller/portal-tools, 2019, accessed: May 2025.

[13] UserLand Software, "XML-RPC specification," [Online]. Available: http://xmlrpc.com/spec.md, 1999, accessed: May 2025.

[14] Pallets Projects, "Flask web framework documentation," [Online]. Available: https://flask.palletsprojects.com/, accessed: May 2025.

[15] Python Software Foundation, “The `xmlrpc.client` module (python 3 documentation),” [Online]. Available: https://docs.python.org/3/library/xmlrpc.client.html, accessed: May 2025.

[16] HashiCorp, “Terraform plugin sdk v2 guidelines,” [Online]. Available: https://developer.hashicorp.com/terraform/plugin/sdkv2, 2023, accessed: May 2025.

[17] kolo, “xmlrpc: A Go XML-RPC client library,” https://github.com/kolo/xmlrpc, 2025, [Online; accessed 18-May-2025].

[18] The Go Authors, “crypto/x509 package,” https://pkg.go.dev/crypto/x509, 2025, [Online; accessed 18-May-2025].