Ατομική Διπλωματική Εργασία

MANAGING CLOUDLAB INFRASTRUCTURE WITH TERRAFORM

Γιάννης Πανής

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Μάιος 2024

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

MANAGING CLOUDLAB INFRASTRUCTRE WITH TERRAFORM

Γιάννης Πανής

Επιβλέπων Καθηγητής

Χάρης Βώλος

Η Ατομική Διπλωματική Εργασία υποβλήθηκε προς μερική εκπλήρωση των απαιτήσεων απόκτησης του πτυχίου Πληροφορικής του Τμήματος Πληροφορικής του Πανεπιστημίου Κύπρου

Μάιος 2024

Acknowledgements

I would like to express my sincere appreciation to my supervisor, Professor Haris Volos, for their guidance and introduction to the topic of my thesis.
I am also incredibly grateful to my family and friends for their support and understanding throughout my studies.

Abstract

The evolution of cloud computing has revolutionized the way organizations deploy, manage, and scale their infrastructure. Terraform, an open-source infrastructure as a code tool, enables users to define and provision data center infrastructure using a declarative configuration language. This thesis presents the design and implementation of a Terraform provider specifically for Cloudlab, a platform for experimenting with cloud computing technologies.

Cloudlab offers a versatile environment for conducting research on cloud infrastructure, but its integration with popular infrastructure as a code tools has been limited. By developing a dedicated Terraform provider for Cloudlab, this work aims to bridge that gap, facilitating automated and efficient management of cloud resources within Cloudlab’s experimental framework.

The development process involved a comprehensive analysis of Cloudlab's API and functionality, followed by the creation of a Terraform provider that supports the full lifecycle of resource management, including provisioning, updating, and decommissioning. Key challenges addressed include ensuring compatibility with Terraform’s architecture, handling authentication and authorization, and maintaining synchronization between Terraform state and Cloudlab’s resource state.

This Terraform provider empowers researchers and developers to leverage Cloudlab’s capabilities with the familiar and powerful tools provided by Terraform, thus enhancing productivity, repeatability, and scalability in cloud computing experiments. The evaluation of the provider demonstrates significant improvements in automation and efficiency, validating its effectiveness and utility in real-world scenarios.

Contents

Chapter 1

Introduction

1.1 Motivation

Infrastructure as a code involves the management and setup of computer data centers using files that computers can understand, instead of manually configuring hardware or using interactive tools. This method enables automation and consistency in infrastructure deployment, making it easier to scale, manage, and maintain complex systems efficiently.

Infrastructure as a code offers a range of advantages that help both employees and business owners. Firstly, it streamlines the process of managing and deploying infrastructure components like servers, networks, and databases. This means employees can set up and modify the infrastructure more quickly and efficiently, saving time and effort.

Moreover, by automating these tasks, infrastructure as a code reduces the likelihood of human error, leading to more reliable and consistent systems. This reliability translates to fewer disruptions in business operations, ensuring smoother workflows and better service delivery to customers.

Additionally, infrastructure as a code facilitates scalability, allowing businesses to easily adapt to changing demands. Whether it is scaling up during peak periods or scaling down to save costs during slower times, the flexibility offered by infrastructure as a code enables businesses to optimize their resources effectively.

Furthermore, by treating infrastructure as a code, businesses can version control their infrastructure configurations. This means they can track changes over time, revert to previous configurations if needed, and ensure compliance with regulatory standards more effectively.
Overall, the adoption of infrastructure as a code results in increased efficiency, reliability, scalability, and compliance, benefiting both employees and business owners alike.

Terraform is an infrastructure as a code software tool created by HashiCorp. Users define and provide data center infrastructure using a declarative configuration language known as HashiCorp Configuration Language (HCL), or optionally JSON[2]. The Terraform tool, commonly used for managing infrastructure as a code, is widely accessible across different cloud platforms like AWS, Azure, and others. Terraform instantiates assets/resources, such as compute, and network components, based on the provided configuration, simplifying the management of infrastructure across various cloud platforms like AWS, Azure, and others.

1.2 Problem

CloudLab is a distributed infrastructure that enables researchers to experiment with cloud computing and networking technologies at a scale. It provides a platform for exploring novel approaches to building and managing clouds and networks, as experiments that terminate after some hours, facilitating research in areas like virtualization, network function virtualization (NFV), software-defined networking (SDN), and edge computing. CloudLab is supported by the National Science Foundation program (NSF) and is operated by a union of universities including the University of Utah, Clemson University, and the University of Wisconsin-Madison. Researchers can access CloudLab resources to deploy custom experiments and testbeds, contributing to advancements in cloud computing and networking research.

As of now, Terraform isn't directly supported on CloudLab, which may limit its usage for university users preferring CloudLab as their cloud service.

Currently there are two ways for managing CloudLab infrastructure. The first and most used way is the graphical user interface on CloudLab website, where user can create or destroy experiments. The other way is through the CloudLab API, which is written in Python. The CloudLab API is not widely used, as it is more complex than graphical user interface on CloudLab website. However, it is used by programmers that want to create mass experiments using Python scripting.

Based on the introduction above, would be helpful to have Terraform work with CloudLab, as students and researchers, would be able to manage the CloudLab infrastructure through Terraform, instead of the graphical user interface or CloudLab API.

Integrating Terraform with CloudLab can offer significant advantages for universities and their users. Terraform's utility lies in its provision of a unified platform for managing cloud infrastructure across various providers. By leveraging Terraform, university users can streamline the deployment and management of resources within CloudLab without the need for specialized interfaces or extensive training on CloudLab's native API. This encourages efficiency by enabling professors to employ a familiar toolset for infrastructure provisioning and also gives the opportunity to students to learn a powerful infrastructure as a code tool like Terraform. Finally this would help researchers who want to manage the CloudLab infrastructure.

CloudLab's unique challenge lies in its focus on managing experiments, which typically involve dynamic and varied network configurations tailored to specific research objectives. Unlike traditional cloud platforms, where infrastructure setup often follows predefined templates, experiments in CloudLab require a more flexible approach due to their diverse networking needs. Experimentation involves deploying and testing various configurations, often without fixed network structures like those found in AWS or Azure, making adaptation of Terraform more complex as it needs to accommodate this fluidity while maintaining infrastructure integrity.

1.3 Objectives

This thesis aims to explore the development of a Terraform provider tailored for CloudLab, leveraging its API to efficiently manage CloudLab infrastructure. A Terraform provider serves as the backbone for Terraform, facilitating interaction with specific cloud services to handle infrastructure operations such as create, read, update and delete.

Our objective is to support resource management practices within CloudLab by preserving the familiar structure of HashiCorp Configuration Language (HCL), commonly utilized for defining infrastructure configurations across various cloud providers. Through this endeavor, we seek to streamline the deployment and management of CloudLab resources, enhancing efficiency and usability for researchers and practitioners alike.

1.4 Contributions

This thesis contributes to the field of cloud infrastructure management by developing a specialized Terraform provider tailored for CloudLab, a versatile cloud computing platform utilized primarily in research and educational settings. The creation of this Terraform provider addresses a critical need within the research community for efficient and standardized infrastructure management tools.

One key contribution lies in the adaptation of Terraform, a widely adopted infrastructure-as-code tool, to seamlessly interface with CloudLab's API. By bridging these technologies, we empower researchers and practitioners to leverage familiar infrastructure management practices while harnessing the capabilities of CloudLab's robust infrastructure.

Also, while the scale-up and scale-down functionality for experiments within CloudLab remains unavailable, the introduction of the specialized Terraform provider offers a pivotal solution to streamline infrastructure management. Researchers and practitioners can now efficiently provision and manage resources on CloudLab using familiar infrastructure-as-code practices facilitated by Terraform. This advancement not only simplifies the process of setting up and configuring environments but also enhances the reproducibility and scalability of experiments conducted within CloudLab.

Furthermore, this thesis enhances the usability and efficiency of CloudLab by providing a unified interface for resource provisioning and management. Through the utilization of HashiCorp Configuration Language (HCL), commonly employed across various cloud providers, users can define and manage CloudLab resources with ease, minimizing the learning curve associated with new tools or platforms.

Another significant contribution lies in the facilitation of streamlined deployment processes within CloudLab. By enabling users to define infrastructure configurations declaratively, researchers can automate the deployment of complex environments, reducing manual intervention and potential errors. This automation fosters reproducibility and scalability, essential aspects of modern research endeavors.

Moreover, this thesis contributes to the broader ecosystem of infrastructure management tools by extending Terraform's capabilities to support CloudLab. As Terraform continues to gain traction as a standard for infrastructure automation, the addition of CloudLab support expands its applicability, benefiting not only researchers but also industry professionals seeking efficient cloud management solutions.

In summary, the development of a specialized Terraform provider for CloudLab presented in this thesis represents a significant contribution to both research and practice. By enhancing the usability, efficiency, and automation capabilities of CloudLab infrastructure management, this work aims to empower users to focus more on their research objectives and less on the intricacies of cloud resource provisioning and management.

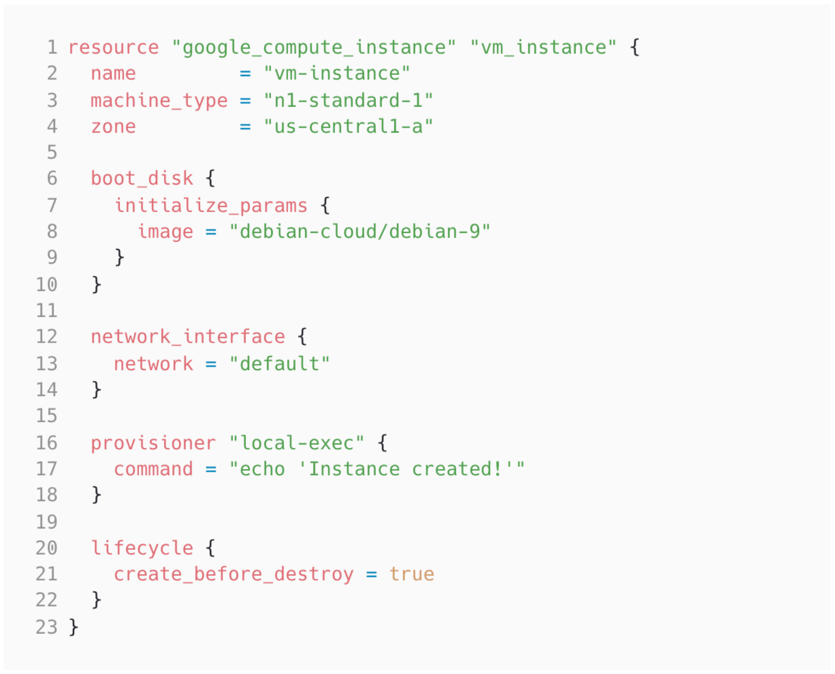
Chapter 2

Background

2.1 Background

2.1.1 Terraform

At its core, Terraform uses a declarative language called HashiCorp Configuration Language (HCL) to define infrastructure resources and their configurations. In this language, you describe the desired state of your infrastructure, such as virtual machines, networks, storage, and more. Terraform then interprets these configurations and figures out what needs to be done to make the actual infrastructure match the desired state.



```hcl
resource "google_compute_instance" "vm_instance" {
  name         = "vm-instance"
  machine_type = "n1-standard-1"
  zone         = "us-central1-a"

  boot_disk {
    initialize_params {
      image = "debian-cloud/debian-9"
    }
  }

  network_interface {
    network = "default"
  }

  provisioner "local-exec" {
    command = "echo 'Instance created!'"
  }

  lifecycle {
    create_before_destroy = true
  }
}
```

Figure 3.2.1.1: Terraform code snippet.

When you initialize Terraform using the command "terraform init," it begins by examining the configuration files in your project to identify any external providers (like AWS, Azure, CloudLab etc.) that it needs to use. Terraform then downloads these modules and plugins, along with any other necessary dependencies, and sets up the environment for managing your infrastructure. This initialization process ensures that Terraform has all the tools and resources it needs to execute your infrastructure code effectively. It's an essential first step before you can start creating, updating, or deleting resources with Terraform.

After the initialization phase, Terraform is now ready to start planning what to create, update or delete. It goes through a process called the "planning phase." During this phase, Terraform examines the configuration files, compares them to the current state of the infrastructure, and determines what actions need to be taken to reach the desired state. It generates an execution plan that outlines these actions, such as creating new resources, updating existing ones, or deleting obsolete ones.

After the planning phase, Terraform can execute the planned actions to make the infrastructure changes. This process is called the "execution phase." Terraform communicates with the APIs of the cloud providers or other infrastructure providers to create, update, or delete the resources as instructed in the execution plan.

Following the execution phase, Terraform proceeds to update its internal state to reflect the changes made to the infrastructure. This internal state serves as a record of the current configuration and status of resources managed by Terraform. By updating this state, Terraform ensures that subsequent operations accurately reflect the existing state of the infrastructure. This step is crucial for maintaining consistency and facilitating future modifications or deployments.

One of the key features of Terraform is its ability to manage dependencies between resources. For example, if you have a web server that depends on a database server, Terraform will ensure that the database server is created first before attempting to create the web server. This helps in building complex infrastructure setups in a reliable and consistent manner.
Overall, Terraform simplifies the management of infrastructure by allowing you to define it in code, providing automation for creating and managing resources, and ensuring consistency and reliability through its dependency management capabilities.

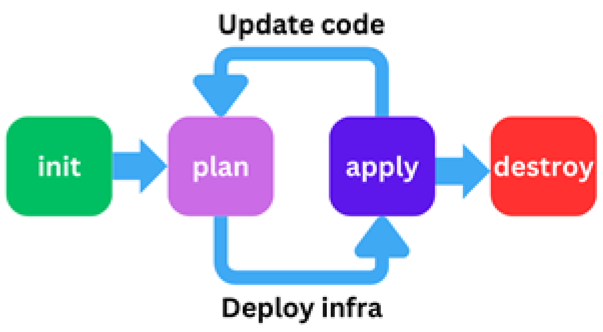


Figure 3.2.1.1: How Terraform works.

Terraform has a framework for writing providers, named "Terraform Plugin Framework". This framework is written in Golang. Also, we will need to use CloudLab API to communicate with CloudLab. This API is written in Python. So, for this Terraform Provider, we will need to integrate Terraform Plugin Framework with CloudLab API.

However, this is not easy, as the CloudLab API and the Terraform Framework Plugin are written in different languages. For that reason, we will need a way to make Terraform Framework Plugin to communicate with CloudLab API. For that reason, we used Flask Framework. Terraform Plugin Framework will send API calls on Flask API, then Flask will forward the request through CloudLab API. Finally, CloudLab API will have some “intelligence”, as it is needed to simulate the network subnet of the resources (Virtual Machines).

2.1.2 Terraform Plugin Framework

To develop a Terraform provider, we will primarily utilize the Terraform Plugin SDK, a comprehensive toolkit designed to facilitate the creation of custom providers for the Terraform infrastructure as a code platform. This SDK offers a set of libraries and tools that streamline the development process, enabling you to interact with various APIs and services programmatically. By harnessing this SDK, developers can seamlessly integrate their infrastructure components into Terraform's ecosystem, enhancing its capabilities to manage diverse resources across different cloud providers and services.

The Terraform plugin framework is an essential component of Terraform, a popular infrastructure-as-code tool used for managing and provisioning infrastructure resources. It enables Terraform to support various providers, such as cloud service providers like AWS, Azure, and Google Cloud Platform, as well as other technologies like Kubernetes, Docker, and databases.

At its core, the plugin framework allows Terraform to communicate with external systems or services through plugins. These plugins extend Terraform's functionality by providing the necessary code to interact with specific providers or technologies. Each plugin serves as a bridge between Terraform and the target system, enabling Terraform to create, modify, and delete resources according to the user's configuration.

The framework follows a modular architecture, where each plugin operates independently of the others. This modular design allows Terraform to support a wide range of providers without tightly coupling them to the core codebase. When Terraform executes a configuration, it loads the necessary plugins based on the resources defined in the configuration. This dynamic loading mechanism ensures that only the relevant plugins are loaded, optimizing performance, and reducing overhead.

Developing a plugin for Terraform involves implementing a set of interfaces defined by the plugin SDK (Software Development Kit). These interfaces include methods for resource management, state management, and configuration validation. Plugin developers write code to handle CRUD operations (Create, Read, Update, Delete) for resources, manage state files, and validate configuration inputs to ensure consistency and reliability.

Once a plugin is developed, it can be distributed and installed separately from Terraform itself. Users can install plugins either manually or through Terraform's built-in plugin installation mechanism. This separation between Terraform's core codebase and plugins allows for easier maintenance and updates, as each plugin can be developed and maintained independently. Additionally, it fosters a vibrant ecosystem where community members can contribute plugins to support new providers or extend Terraform's capabilities. Overall, the Terraform plugin framework plays a crucial role in enabling Terraform's flexibility, extensibility, and interoperability with various infrastructure technologies.

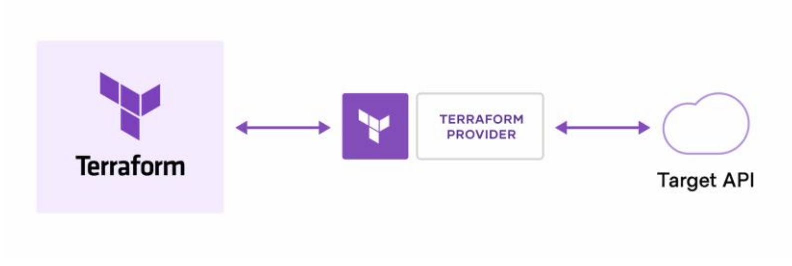


Figure 2.1.2.1: How Terraform Providers work.

To develop a plugin for Terraform, you start by defining the functionality you want to add, such as support for a new infrastructure provider or extending Terraform's capabilities. Then, you create the plugin code using the appropriate programming language and follow Terraform's plugin development guidelines, ensuring compatibility and adherence to best practices. Once the plugin is developed and tested, it can be packaged into a distributable format, typically a binary file, along with any necessary documentation.

Distributing the plugin involves making it available for users to install either manually or through Terraform's built-in plugin installation mechanism. This can be done by hosting the plugin file on a public repository or marketplace, such as the Terraform Registry, or by providing direct download links. Users can then install the plugin by following the installation instructions provided, enabling them to leverage the new functionality within their Terraform workflows. This separation between Terraform's core codebase and plugins allows for easier maintenance and updates, fostering a vibrant ecosystem of community-contributed plugins that enhance Terraform's flexibility and interoperability with various infrastructure technologies.

2.1.3 Cloudlab

Thus far, there has been a notable absence of infrastructure-as-code capabilities for CloudLab. Currently, CloudLab offers a Python-based API enabling users to initiate, terminate, and establish connections between experiments. Nonetheless, a structured system for managing infrastructure through code remains unavailable within the CloudLab framework. This absence presents an opportunity to enhance CloudLab's functionality by incorporating infrastructure as a code methodologies, thereby streamlining experiment management and deployment processes. Expanding CloudLab's capabilities in this manner would empower users to automate and orchestrate their experiments more efficiently, fostering greater flexibility and scalability within the platform. Although we don’t want to give access to anyone through an infrastructure as a code tool. Only users with existing access will be able to use Terraform to manage CloudLab infrastructure.

Upon instantiation of an experiment, the profile code springs into action, executing its predefined instructions to generate an XML code. This XML code serves as a vital directive for CloudLab, guiding its automated processes in creating the experiment environment precisely as specified. Through this systematic approach, the profile ensures consistency and accuracy in setting up experiments, laying the groundwork for seamless execution and reliable results within the CloudLab infrastructure.

2.2 Related Work

In the context of managing cloud infrastructure, various tools and approaches have been developed to streamline and optimize the deployment, configuration, and maintenance of resources. Terraform by HashiCorp is one such tool that has gained widespread adoption due to its declarative approach and support for multi-cloud environments. This subchapter explores related work in this domain, highlighting tools and frameworks that share similar objectives with Terraform, particularly in the context of managing cloud infrastructure like CloudLab.

One of the primary alternatives is AWS CloudFormation. CloudFormation is a service provided by AWS (Amazon Web Services), that works with JSON and YAML files. However, it integrates only with AWS services. This can be a limitation for multi-cloud environments, where Terraform offers more flexibility.

Another notable tool is Ansible provided by Red Hat. Ansible works with YAML files, making it accessible for users already familiar with its syntax. However, unlike Terraform, Ansible is designed to handle both the initial setup and ongoing configuration management, making it more complex when focusing only on infrastructure provisioning.

Also, another tool is Enoslib. Enoslib is a Python library designed to facilitate the deployment and management of experimental infrastructures. It is particularly useful for researchers and engineers who need to set up complex and reproducible experimental environments. Enoslib supports various cloud and cluster environments, including those orchestrated by OpenStack, Grid5000, and Docker. It offers a higher-level abstraction compared to Terraform, focusing on ease of use for experimental deployments.

These tools highlight the ecosystem of infrastructure as a code solution, each with its strengths and weaknesses. While Terraform is a highly popular and flexible choice due to its multi provider support and declarative syntax, understanding the alternatives allows for a more informed decision when selecting the most suitable tool for each case.

Also there are several tools analogous to CloudLab, each offering different capabilities. One of them is OpenStack, an open-source cloud computing platform. Another significant tool is Google Cloud Platform by Google, which offers a comprehensive suite of cloud computing services. Finally, and the pioneer of the cloud computing is AWS (Amazon Web Services) which offers a variety of cloud computing services. All of them are used in both academic and industry. However, none of these services provide the “experiments” that CloudLab has.

Chapter 3

Architecture

3.1 Terraform Provider

The Terraform Provider plays a crucial role in ensuring that users provide the right parameters and promptly communicates any errors or successes. Additionally, it verifies the presence of a valid credentials .pem file. While it doesn't validate the correctness of the credentials, it does confirm that the .pem file exists and that Terraform has the necessary permissions to access it. This helps maintain the integrity of the configuration process and aids in smooth execution of Terraform operations.

3.2 Intermediary API

Intermediary API is responsible for transmitting experiment parameters and authentication credentials (stored in a .pem file) to the Cloudlab API. It also acts as a state management service, keeping track of the state, information if they exist and which experiment created them, for all VLANs created by the Terraform provider. This ensures that no experiment gets deployed into a non-existent VLAN.

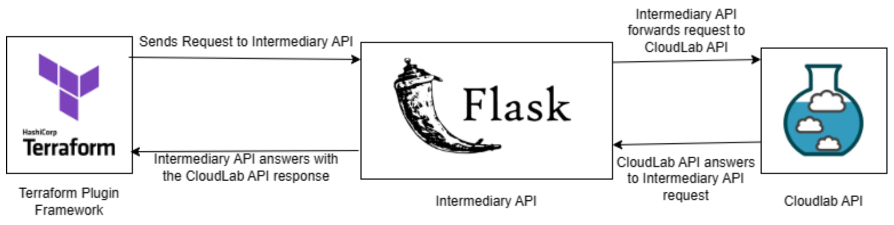
With the integration of the Terraform provider and Intermediary API, the capability to scale up and down experiments, which was previously unavailable within Cloudlab, has now become accessible. Through this enhanced infrastructure, researchers and developers can dynamically adjust the scale of their experiments, optimizing resource utilization and facilitating more efficient testing and deployment processes. This advancement marks a significant evolution in experiment management within Cloudlab, empowering users with greater flexibility and control over their computational environments.

In addressing the challenge of scaling experiments up and down efficiently, a solution is to use VLANs provided by CloudLab. VLANs enable experiments within the same network to communicate. This allows for more flexible experimentation as researchers can allocate resources dynamically, scaling their setups according to changing requirements. With VLANs, CloudLab helps users control their experiments better, making it easier to switch between different experiment sizes smoothly.

Another reason for the necessity of an intermediary API, except the VLAN state maintain, lies in the divergence of programming languages between systems. In this case, the CloudLab API is coded in Python, while the Terraform Plugin Framework operates on Go. These distinct programming languages pose a challenge for direct communication and integration between the two systems. To bridge this linguistic gap and enable seamless interaction, an intermediary API serves as a mediator, facilitating communication by translating requests and responses between Python and Go. By providing a unified interface that abstracts the underlying language complexities, the intermediary API fosters interoperability, allowing the CloudLab API and Terraform Plugin Framework to effectively collaborate despite their disparate linguistic foundations. This intermediary layer not only harmonizes communication but also enhances system compatibility and scalability, thereby streamlining development and deployment processes in heterogeneous software environments.

3.3 Putting it all together

To summarize, the Terraform provider sends an API call with the parameters. The Flask API checks if any of the VLANs of the node requested already exists in another experiment. If any of the VLANs requested already exist, it checks if the experiment that created the each VLAN has passed the provisioning state. If not, then it waits until it passes the provisioning state. If any of the VLAN is already created, it will connect the requested node into it, otherwise it will create it using the process we explained before. After the ‘VLAN checks’ phase, it forwards the request to CloudLab API. It gets the response from the CloudLab API, it returns the appropriate response to Terraform provider. It’s worth noting that before any checks, Flask API checks if the correct arguments are given by Terraform provider. If not, it returns the appropriate error to Terraform provider.


Sends Request to Intermediary API
Intermediary API forwards request to CloudLab API
HashiCorp Terraform
Flask
Intermediary API answers with the CloudLab API response
CloudLab API answers to Intermediary API request
Terraform Plugin Framework
Intermediary API
Cloudlab API

Chapter 4

Terraform Provider

4.1 How Terraform Works

Terraform utilizes HashiCorp Configuration Language (HCL) to define infrastructure resources and configurations, aiming to achieve a desired state of infrastructure. After initialization using the "terraform init" command, Terraform enters a planning phase where it analyzes configuration files and generates an execution plan to reconcile the desired state with the current infrastructure. Subsequently, Terraform executes the planned actions, communicating with cloud or infrastructure providers' APIs to create, update, or delete resources accordingly. Throughout this process, Terraform manages dependencies between resources to ensure consistency and reliability in infrastructure deployment. This workflow underscores Terraform's role in simplifying infrastructure management through code-based definitions, automation, and dependency management capabilities.

4.2 Terraform Plugin Framework

Developing a plugin for Terraform involves implementing a set of interfaces defined by the plugin SDK (Software Development Kit). These interfaces include methods for resource management, state management, and configuration validation. Plugin developers write code to handle CRUD operations (Create, Read, Update, Delete) for resources, manage state files, and validate configuration inputs to ensure consistency and reliability.

4.2.1 Terraform Provider Publish

Publishing a Terraform plugin involves several key steps to ensure that your provider is reliable, secure, and available to the community. The process starts with preparing your plugin for release, which includes generating a GPG signing key, verifying various configurations, and creating a release on GitHub. Following these steps helps maintain the integrity of your plugin and facilitates a smooth publishing process.

Github Actions are used to automate workflows, including building, testing, and releasing your provider. Verifying your GitHub Action workflow involves ensuring that your CI/CD pipeline is correctly configured to handle these tasks. This step includes checking for proper triggers, dependencies, and steps in the workflow file, which helps in automating the validation and release process, ensuring consistency and reliability.

A GPG signing key is used to sign your provider's release artifacts, ensuring their authenticity and integrity. Generating a GPG signing key involves creating a new key pair that you will use to sign your releases. This step is critical for security, as it allows users to verify that the releases they download are indeed from you and have not been tampered with.

Creating a provider release involves tagging a new version in your GitHub repository and pushing it. This action triggers your GitHub Action workflows, which use GoReleaser to build your provider binaries, sign them with your GPG key, and publish them as a new release on GitHub. This step consolidates all your preparations and automation efforts, resulting in a new version of your provider being available for users.

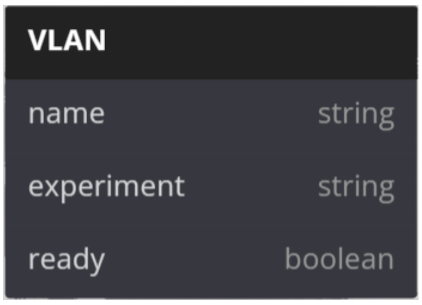
Chapter 5

Intermediary API

In this chapter, we delve into the mechanics of Flask API and how we used it to achieve the desired result. This is a crucial component bridging the functionalities of Terraform provider and CloudLab API. Flask serves as a pivotal intermediary, facilitating seamless communication between these two entities. Its primary role encompasses the forwarding of API calls, adeptly managing errors, and preserving VLANs state.

5.1 SQLite

Using SQLite as my relational database to store the VLANs states inside. The schema for this database is a simple format using SQL statements to describe and manage the tables of the following schema.



VLAN	
name	string
experiment	string
ready	boolean

Figure 3.2.1.1: Flask API Database Schema.

The VLAN table comprises the following columns:

- Name: This refers to the name given to the VLAN.
- Experiment: Denotes the specific experiment responsible for the VLAN's creation.
- Ready: Indicates whether the experiment associated with the VLAN has reached a state after provisioning, denoted by a value of 'True'. If the experiment is not yet ready, it is indicated by a value of 'False'. Before provisioning of the experiment, the VLAN was not created yet, so is not usable.

The intermediary Flask API, tasked with managing VLAN states via a database table, operates with a pivotal function: preserving the status of each VLAN. This entails that when the API receives a request to retrieve the experiment status linked to a specific VLAN, it undertakes the crucial responsibility of updating the corresponding database records. This dynamic process ensures that any alterations in the experiment's status, such as termination, prompt the API to promptly delete the associated record pertaining to the VLAN involved in the experiment. By executing these operations seamlessly, the API maintains an accurate and up-to-date representation of VLAN states, thereby facilitating efficient management and oversight of network configurations.

5.2 Flask

A Flask API is a web application programming interface (API) built using Flask, a micro web framework for Python. An API is a way for two components to communicate with each other. API takes requests from clients (like web browsers or mobile apps), processes them, and returns responses. Flask makes it easy to create these APIs by providing tools and libraries to handle HTTP requests and responses.

In a Flask API, you define routes, which are URLs that clients can visit to interact with your API. For example, you might have a route like "/hello" that responds with a friendly greeting when visited. Each route is associated with a function called a view function, which runs when the route is visited. Inside these view functions, you can perform tasks like fetching data from a database, processing user input, or returning specific responses.

Flask APIs are commonly used for building web services, backend systems, and microservices. They're flexible and lightweight, making them a popular choice for developers who want to quickly create APIs without a lot of overhead. With Flask, you can easily build APIs for tasks like serving data to a web or mobile application, handling webhook requests from third-party services, or even building your own custom web services.

It's crucial to emphasize that to access the CloudLab API, users must possess a credentials .pem file downloadable from their account. Consequently, the Flask API necessitates the submission of this credentials .pem file to facilitate communication with the CloudLab API.

A .pem file, short for Privacy-Enhanced Mail, is a file format used to store cryptographic keys and certificates. It typically contains either a public key, a private key, or a certificate issued by a certificate authority. These files are commonly used in secure communication protocols like SSL/TLS to establish encrypted connections over networks such as the internet.

To utilize the CloudLab API, users must acquire the necessary credentials in the form of a .pem file, which can be downloaded from the platform. However, upon downloading, the file is encrypted, necessitating decryption before use. This decryption process involves employing specific commands using the OpenSSL tool. Specifically, the commands "openssl rsa -in cloudlab.pem" and "openssl x509 -in cloudlab.pem" are utilized to decrypt the file, resulting in a decrypted .pem file termed as "cloudlab-decrypted.pem". Once decrypted, this file can be used to authenticate and access the CloudLab API securely, enabling users to leverage its services and resources effectively.

5.3 Deploy on Docker

We decided to deploy the API using Docker primarily due to its user-friendly nature. Utilizing Docker simplifies the process for users, they just need to clone the API repository and execute the Docker Compose file. This approach eliminates the complexities typically associated with setting up and configuring the API environment. By leveraging Docker, users can quickly and effortlessly establish a consistent and reliable deployment environment, enhancing overall efficiency and ease of use.

Except of the ease of user, deploying the API using Docker offers a plethora of advantages that streamline the development process. Docker provides a consistent environment across different platforms, ensuring the seamless execution of the API regardless of the underlying infrastructure. This consistency mitigates compatibility issues and simplifies deployment, as developers can package all dependencies and configurations within Docker containers. Additionally, Docker's lightweight nature optimizes resource utilization, enhancing performance while minimizing overhead costs. Overall, leveraging Docker for API deployment enhances reliability and efficiency, making it an indispensable tool for modern software development practices.

5.4 Testing

Testing the API posed a significant challenge due to its requirement for a .pem file as a parameter, a feature unsupported by standard API testing tools. Consequently, the conventional tools were inadequate for assessing the API's functionality accurately. To overcome this limitation, a bespoke solution was developed in Go programming language. This custom Go program was specifically crafted to facilitate the testing process, enabling the seamless transmission of the necessary .pem file to the API for comprehensive evaluation. Through the utilization of this tailored approach, the testing endeavor was effectively streamlined, ensuring thorough assessment of the API's capabilities and performance.

Chapter 6

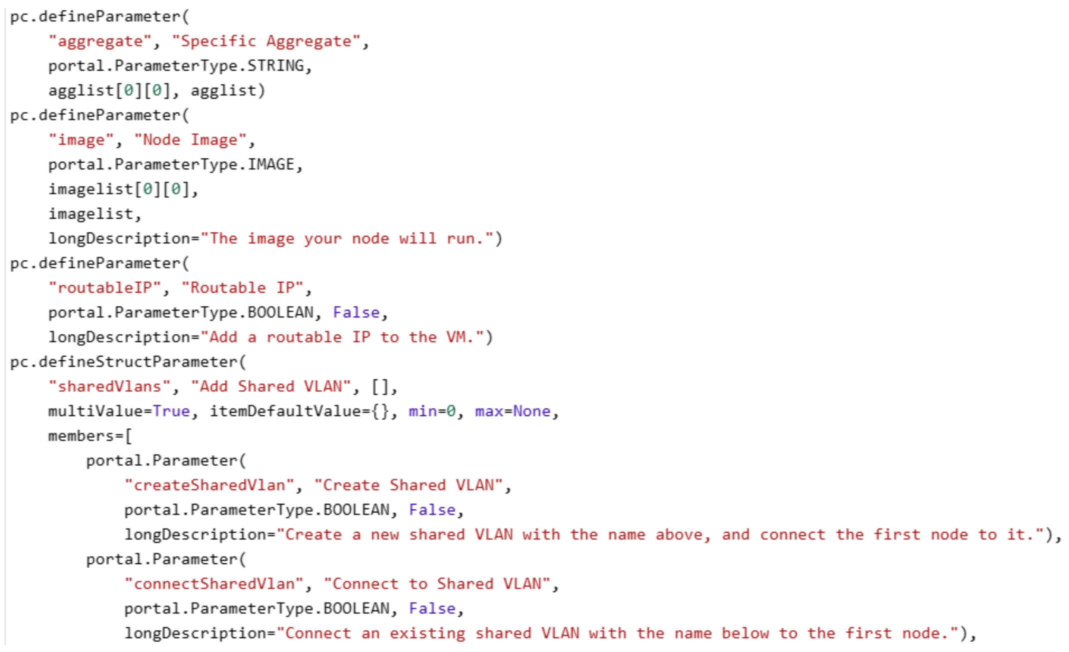
Cloudlab Profile

3.1 Cloudlab Profile

For this reason, we created a profile named Terraform-profile in the UCY-COAST project. We made this profile accessible to everyone. The profile instantiates a single node and gives the possibility to the user to connect it to a VLAN. This profile takes as parameters:

- Specific Aggregate: The servers where the node will be. Users can choose between emulab.net, utah.cloudlab.us, clemson.cloudlab.us, wisc.cloudlab.us, apt.emulab.net, Any.
- Node Image: The operating system the node will have. Users can choose between Ubuntu 18.04, Ubuntu 16.04, Ubuntu 20.04, Centos 7, FreeBSD 11.3.
- Routable IP: A checkbox, if it is checked, then the node is accessible from outside the CloudLab network.

- Shared VLANs: A list of VLANs the node will belong to. Each VLAN have the following fields:
    - Create Shared VLAN: A checkbox, if true then the VLAN with name given will be created by this node.
    - Connect Shared VLAN: A checkbox, if true then the node will use the existing VLAN with name given.
    - Shared VLAN Name: The name of the VLAN.
    - Shared VLAN IP Address: The IP Address that the node will have in the VLAN.
    - Shared VLAN Netmask: The Subnet Mask that the node will have in the VLAN.



```
pc.defineParameter(
    "aggregate", "Specific Aggregate",
    portal.ParameterType.STRING,
    agglist[0][0], agglist)
pc.defineParameter(
    "image", "Node Image",
    portal.ParameterType.IMAGE,
    imagelist[0][0],
    imagelist,
    longDescription="The image your node will run.")
pc.defineParameter(
    "routableIP", "Routable IP",
    portal.ParameterType.BOOLEAN, False,
    longDescription="Add a routable IP to the VM.")
pc.defineStructParameter(
    "sharedVlans", "Add Shared VLAN", [],
    multiValue=True, itemDefaultValue={}, min=0, max=None,
    members=[
        portal.Parameter(
            "createSharedVlan", "Create Shared VLAN",
            portal.ParameterType.BOOLEAN, False,
            longDescription="Create a new shared VLAN with the name above, and connect the first node to it."),
        portal.Parameter(
            "connectSharedVlan", "Connect to Shared VLAN",
            portal.ParameterType.BOOLEAN, False,
            longDescription="Connect an existing shared VLAN with the name below to the first node."),
```

3.2 Profile Auto Update

The profile operates on a repository basis, indicating that the source code of the profile is stored within a git repository. Additionally, a webhook has been integrated into the repository, allowing for the automatic triggering of a POST request to CloudLab whenever a new push occurs. This facilitates the seamless updating of the profile on CloudLab, ensuring its continuous alignment with the latest changes made to the source code.

3.3 Multi-Node Deployment for Distributed Computing Usage

In a distributed computing environment, the deployment of multiple nodes plays a pivotal role in harnessing computational power effectively. The Terraform-profile simplifies this process by providing a streamlined approach to configure and deploy nodes across cloud infrastructures. Let's delve into an example scenario to illustrate its application.

Imagine a research project requiring a distributed computing environment for analyzing vast datasets. With the Terraform-profile, researchers can define configurations for creating multiple nodes, each tasked with specific computational roles. For instance, one node may serve as a master node orchestrating tasks, while others act as worker nodes processing data in parallel.

The Terraform-profile offers flexibility in selecting node images tailored to the project's requirements. Researchers can opt for operating systems conducive to distributed computing frameworks like Kubernetes or Hadoop. These frameworks enable seamless management of distributed applications, allowing for efficient resource utilization and fault tolerance.

Integration with tools such as Kubernetes enhances the scalability and resilience of the distributed computing environment. Kubernetes automates the deployment, scaling, and management of containerized applications across clusters of nodes. By leveraging the Terraform-profile, researchers can provision Kubernetes clusters with predefined configurations, ensuring consistency and reproducibility.

Similarly, Hadoop facilitates distributed data processing by distributing datasets across multiple nodes and parallelizing computation tasks. With the Terraform-profile, researchers can provision Hadoop clusters effortlessly, configuring nodes with appropriate roles such as NameNode, DataNode, and ResourceManager.

One of the significant advantages of using the Terraform-profile is its ability to extend or decrease the number of nodes even after the initial creation. This dynamic scalability ensures that the computing environment can adapt to varying workloads and computational demands. Researchers can easily add more nodes to handle increased data processing needs or reduce the number of nodes to optimize resource usage and manage costs effectively. This can be done by creating another experiment using the Terraform-profile and connecting it to VLANs that have been created by another profile. This interconnected setup allows for flexible resource allocation and ensures that all nodes can communicate seamlessly within the same network, further enhancing the efficiency and scalability of the distributed computing environment.

In this way, the Terraform-profile serves as a foundational tool for establishing robust distributed computing environments, empowering researchers to tackle complex computational challenges effectively. By integrating seamlessly with frameworks like Kubernetes and Hadoop, it facilitates the implementation of scalable and resilient solutions for distributed data processing and analysis.

Chapter 7

Limitations

In this chapter, we delve into the various limitations encountered when developing Terraform Provider. Understanding these limitations is essential for effectively navigating the intricacies of each tool and optimizing their usage within specific contexts.

CloudLab, as a cloud infrastructure platform, imposes certain constraints that users must be aware of when provisioning and managing their resources. These restrictions range from limitations on virtual machine configurations to constraints on the number of failed experiments allowed. By exploring these CloudLab restrictions, we gain insights into the challenges users may face during the instantiation of resource.

Similarly, Terraform, a powerful infrastructure as a code tool, comes with its own set of limitations and constraints. These Terraform restrictions may include limitations on resource types, dependencies, or scalability. By examining these restrictions, we can identify potential roadblocks in infrastructure provisioning workflows and explore workarounds or alternative approaches to address them effectively.

Furthermore, the Go programming language, commonly used for developing Terraform providers and other infrastructure-related tools, presents its own unique set of restrictions. These Go restrictions were mostly language features. Understanding these constraints is essential for writing efficient and reliable code in Go and ensuring compatibility with Terraform and other infrastructure management tools.

By dissecting the restrictions associated with CloudLab, Terraform, and Go, this chapter aims to provide a comprehensive understanding of the limitations inherent in each tool. Armed with this knowledge, users can better navigate the complexities of infrastructure management and develop strategies to overcome obstacles encountered during the provisioning and maintenance of cloud resources.

4.1 CloudLab Limitations

CloudLab, while offering valuable infrastructure management capabilities, imposes certain restrictions that users must navigate. Understanding and working within these limitations is crucial for effectively utilizing CloudLab for infrastructure provisioning and management.
However, one notable limitation is the inability to modify experiments programmatically through the Cloudlab API. While Cloudlab provides a user-friendly web interface for conducting various tasks such as creating, starting, stopping, and deleting experiments, there is currently no direct support for experiment modification via API endpoints. This limitation restricts automation and integration possibilities for users who prefer to manage experiments programmatically or through scripting.

During my experience developing the Terraform provider, I encountered a notable restriction concerning the handling of repeated failed experiments on CloudLab. Whenever a user attempted to create an experiment that failed consistently for the same reason, CloudLab would freeze the user's account. This necessitated contacting support to resolve the issue, thereby imposing a limitation on the number of failed experiments a user could initiate before facing account suspension.

This constraint notably affected certain approaches, such as the creation of VLANs, where multiple failed attempts were common due to issues like VLAN non-existence. Implementing solutions to automatically retry failed experiments, such as attempting to create a VLAN with parameters to connect to it and creating it if it did not exist, was hindered by the restriction on the number of failed experiment attempts allowed per user.

Navigating these restrictions requires careful consideration and potentially alternative approaches to achieve desired infrastructure configurations within CloudLab. Awareness of these limitations empowers developers to develop strategies that optimize resource utilization and mitigate potential disruptions to their infrastructure management workflows.

4.2 Terraform Limitations

While Terraform offers powerful capabilities for infrastructure management, it also comes with certain restrictions that can impact the development process of Terraform providers. One significant limitation arises from the language in which Terraform providers are written. Terraform providers are typically developed in Go, whereas CloudLab, the infrastructure being managed, is written in Python. This disparity presents a challenge for seamless integration, as one approach considered was to incorporate Python files within the Terraform provider and execute them using bash commands. However, the Terraform plugin framework only downloads Go files when utilizing the provider, making it impossible to execute Python code within the Terraform provider environment.

4.3 Go Limitations

In addition to its strengths, Go also presents certain limitations, one of which poses a significant challenge in interfacing with external systems like the CloudLab API. The CloudLab API, written in Python, communicates with CloudLab via XMLRPC (XML Remote Procedure Calling) client and invokes XMLRPC functions. However, Go lacks an official library from Google that fully supports all the functionalities of Python's XMLRPC.
This absence of comprehensive XMLRPC support in Go necessitated the development of an intermediary API written in Python. This intermediary API serves as a bridge between the Terraform provider, written in Go, and the CloudLab API. Essentially, the Terraform provider interacts with the intermediary API, which, in turn, communicates with the CloudLab API through Python functions.

The requirement for an intermediary API arises from the disparity between the capabilities of Python's XMLRPC libraries and the limitations of existing Go libraries. While Go offers robust performance and concurrency features, its ecosystem may lag behind in certain areas compared to more established languages like Python. Thus, the intermediary API acts as a workaround to leverage the functionalities of Python and facilitate seamless communication between the Terraform provider and the CloudLab API.

Despite this restriction, the integration of an intermediary API enables the Terraform provider to effectively interact with the CloudLab infrastructure, thereby extending the capabilities of Go-based applications in managing cloud resources. Moving forward, efforts to address this limitation may involve exploring alternative approaches or libraries within the Go ecosystem or advocating for enhancements to Go's XMLRPC support to better align with the requirements of interfacing with external systems like CloudLab.

Chapter 8

Evaluation

8.1 Terraform vs CloudLab Web UI

The Terraform vs CloudLab Web UI chapter serves as a critical examination of the Terraform approach compared to the Web UI of CloudLab in managing infrastructure. Each method presents distinct advantages and limitations, which warrant careful consideration for users seeking the most suitable approach for their needs.

8.1.1 Ease of Use

In evaluating Terraform versus CloudLab Web UI for ease of use, several factors were considered. CloudLab Web UI provides a graphical user interface (GUI) that gives the potential of provisioning and managing cloud resources. Users can interact with intuitive menus and forms, reducing the need for extensive coding knowledge. This accessibility enhances usability, especially for those less experienced in infrastructure as a code practices.

On the other hand, Terraform, a tool for building, changing, and versioning infrastructure safely and efficiently, offers a command-line interface that requires familiarity HashiCorp configuration language files. Users must have a solid understanding of infrastructure as a code principles to effectively utilize Terraform. Moreover, Terraform's uses extend further than a single cloud provider; it can be used across various platforms such as AWS, Azure, and Google Cloud. This cross-compatibility means that learning Terraform for one cloud provider can be used for others as well, streamlining the learning process and enhancing its value proposition.

8.1.2 Repeatability

Repeatability refers to the ability to reliably recreate infrastructure configurations. Terraform excels in this aspect due to its declarative approach. Infrastructure configurations are defined in code, ensuring consistency across deployments. By versioning configuration files and leveraging infrastructure state management, Terraform enables precise replication of environments.

CloudLab Web UI offers a different approach to repeatability. It maintains a history of experiments created by users. Each experiment in the history has a button associated with it that allows users to recreate the same experiment with a single click. This feature streamlines the process of reproducing previous setups. However, it's important to note that CloudLab Web UI lacks built-in versioning capabilities. While users can easily recreate experiments, there is no mechanism for tracking and managing changes over time.

8.1.3 Functionality

Terraform offers a streamlined and automated approach to infrastructure management through its declarative configuration files. With Terraform, users can define their desired infrastructure state in code, enabling easy replication and version control. This approach promotes consistency and reproducibility across environments, reducing the likelihood of configuration errors and ensuring that infrastructure deployments are predictable and reliable. Additionally, Terraform's support for infrastructure as a code principles facilitates collaboration among team members and simplifies the process of scaling infrastructure as workloads grow.

In contrast, the Web UI of CloudLab provides a graphical interface for managing infrastructure, offering a more intuitive and visually oriented experience. This interface may appeal to users who prefer a point-and-click approach to infrastructure management or who are less familiar with coding and infrastructure as a code concepts. The Web UI offers real-time feedback and visualizations, allowing users to interactively explore and configure their infrastructure components. However, reliance on the Web UI may introduce challenges in terms of reproducibility and automation, as manual configuration steps are susceptible to human error and may be difficult to track and replicate consistently across environments. Additionally, the graphical nature of the interface may limit the scalability and complexity of infrastructure configurations that can be effectively managed through the Web UI alone.8

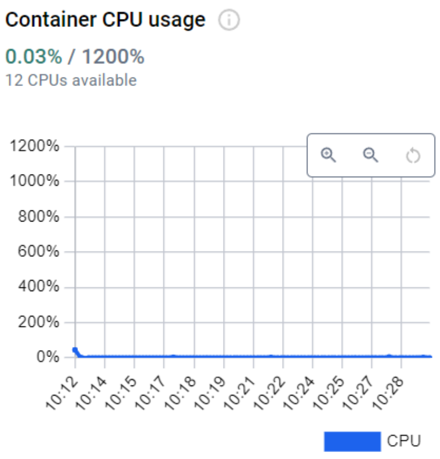
8.2 Performance Evaluation

When the intermediary API receives requests to create virtual machines, it serializes the process at a specific point. This point occurs when a virtual machine needs to create the VLAN before the other virtual machines can connect to it. At this juncture, there is a delay of about 10 seconds while the first VM creates the VLAN. Following this initial delay, the creation of subsequent VMs proceeds in parallel. This serialization ensures that the network configuration is correctly established before other virtual machines are instantiated, thus avoiding potential connectivity issues. However, this initial delay introduces a slight overhead to the provisioning process. Optimizing this aspect of the API or exploring alternative approaches to VLAN creation could further enhance the performance and efficiency of the system.

8.2.1 CPU and Memory Utilization

The plots below show the CPU and Memory Utilization while creating 7 VMs that are connected to the same 2 VLANs. As you can see, both CPU and Memory Utilization is close to 0%.

The minimal resource utilization indicates that the intermediary API does not impose significant computational or memory load during the VM creation process. This efficiency suggests that the API's performance is primarily limited by the VLAN creation step rather than by the computational overhead. Future improvements could focus on optimizing the VLAN creation to further reduce delays and enhance the overall provisioning speed without compromising resource efficiency.



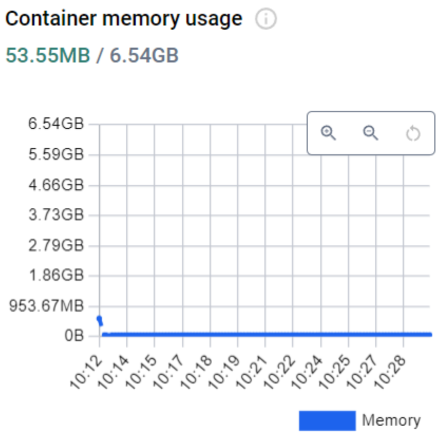


Figure 8.2.1.1 CPU and Memory Utilization while creating 7 VMs connected to 2 VLANs

Chapter 9

Conclusion

6.1 Conclusions

In conclusion, this thesis has explored the development and implementation of a Terraform provider for managing CloudLab infrastructure, specifically focusing on the creation and deletion of virtual machines. Through this research, it has been demonstrated that utilizing Terraform as an infrastructure management tool offers significant advantages in terms of automation, scalability, and reproducibility.

By creating a custom Terraform provider tailored to CloudLab, users can efficiently provision and manage virtual machines within their infrastructure environment. This not only streamlines the process of deploying resources but also enhances flexibility and control over the infrastructure's configuration.

Furthermore, this project contributes to the broader field of cloud computing by extending the capabilities of Terraform to support diverse cloud platforms and environments. By leveraging Terraform's declarative syntax and infrastructure as a code principles, users can easily define and manage complex infrastructure setups with minimal effort.

Overall, the development of a Terraform provider for CloudLab represents a significant step towards enhancing the efficiency and reliability of cloud infrastructure management. This research opens avenues for further exploration and innovation in the realm of automated infrastructure provisioning, paving the way for more seamless and scalable cloud deployments in the future.

6.2 Future Work

Moving forward, several avenues for future work emerge from this thesis's findings and implementation. Firstly, enhancing the Terraform provider to include an Update functionality stands as a promising direction. Currently, the provider focuses on creating and destroying virtual machines, but integrating an Update feature would enable users to modify existing resources without the need for manual intervention. This would not only streamline the management process but also ensure that infrastructure configurations remain up-to-date and adaptable to evolving requirements.

Additionally, removing the intermediary Flask API presents another opportunity for refinement. While the Flask API served as a bridge between Terraform and CloudLab in the current implementation, eliminating this layer could simplify the architecture and reduce potential points of failure. By directly integrating CloudLab functionalities into the Terraform provider, users can benefit from a more seamless and efficient workflow, bypassing unnecessary layers and enhancing overall system performance.

Moreover, extending the Terraform provider's capabilities beyond virtual machines holds promise for advancing infrastructure management. For instance, incorporating support for additional CloudLab resources like networks, storage volumes, or security groups could furnish users with a more comprehensive toolkit for provisioning and configuring their cloud infrastructure. Continual refinement and expansion of the Terraform provider by researchers and practitioners stand to enrich cloud infrastructure management practices, fostering greater efficiency, scalability, and reliability in cloud deployments. This expansion might entail modifying the profile settings on CloudLab to encompass a broader array of parameters. Through such modifications, users could gain the ability to create diverse types of virtual machines, thus broadening the utility and flexibility of the infrastructure provisioning process.

Currently, the intermediary API holds the state of VLANs on a database locally, which restricts users to working on a single machine to access the same state. A potential future enhancement could address this limitation by enabling a distributed state management system. This would allow users to work on different computers while maintaining consistent access to the same state. Implementing such a solution would improve collaboration and flexibility, as users would no longer be tethered to a specific machine to manage VLAN configurations, thereby enhancing the overall usability and robustness of the system.

References

[1] https://en.wikipedia.org/wiki/Infrastructure_as_code

[2] https://en.wikipedia.org/wiki/Terraform_(software)

[3] https://developer.hashicorp.com/terraform/plugin/framework

[4] https://spacelift.io/blog/ansible-vs-terraform

[5] https://flask.palletsprojects.com/en/latest/api/

[6] https://www.sqlite.org/

[7] https://www.terraform.io/

[8] https://zeet.co/blog/terraform-alternatives

[9] https://docs.cloudlab.us/getting-started.html

[10] https://gitlab.flux.utah.edu/stoller/portal-tools

[11] https://www.usenix.org/conference/atc19/presentation/duplyakin

[12] https://aws.amazon.com/cloudformation/

[13] https://github.com/BeyondTheClouds/enoslib

[14] https://discovery.gitlabpages.inria.fr/enoslib/

[15] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The design and operation of cloudlab. In Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '19). USENIX Association, USA, 1–14.

[16] R. -A. Cherrueau et al., "EnosLib: A Library for Experiment-Driven Research in Distributed Computing," in IEEE Transactions on Parallel and Distributed Systems, vol. 33, no. 6, pp. 1464-1477, 1 June 2022

[17] https://github.com/BeyondTheClouds/enoslib

[18] https://hal.science/hal-03324177/

[19] https://ieeexplore.ieee.org/abstract/document/9139623/