Enterprise SaaS Workloads on New-Generation Infrastructure-as-Code (IaC) on Multi-Cloud Platforms

Sandesh Achar

Director of Cloud Engineering, Workday Inc., Pleasanton, California, **USA**

ABSTRACT

Cloud Computing has become the primary model used by DevOps practitioners and researchers to provision infrastructure in minimal time. But recently, the traditional method of using a single cloud provider has fallen out of favor due to several limitations regarding performance, compliance rules, geographical reach, and vendor lock-in. To address these issues, industry and academia are implementing multiple clouds (i.e., multi-cloud). However, managing the infrastructure provisioning of enterprise SaaS applications faces several challenges, such as configuration drift and the heterogeneity of cloud providers. This has seen Infrastructure-as-Code (IaC) technologies being used to automate the deployment of SaaS applications. IaC facilitates the rapid deployment of new versions of application infrastructures without degrading quality or stability. Therefore, this work presents a vision of uniformly managing the infrastructure provisioning of enterprise SaaS applications that utilize multiple cloud providers. Hence, we introduce an initial design for the IaC-based Multi-Cloud Deployment pattern and discuss how it addresses the relative challenges.

Key Words: Infrastructure Provisioning, SaaS, Infrastructure as Code, Cloud Computing, Multi-Cloud

INTRODUCTION

In most enterprises, the biggest challenge is delivering a new idea or software feature to customers as quickly as possible. Unfortunately, slow and error-prone manual workflows, slow ticketing systems, and unscalable infrastructure can drastically slow down the software release lifecycle. To become more agile, enterprises use DevOps (Development & Operations) to shorten the software development lifecycle. DevOps refers to a set of practices and tools that promote continuous collaborations between developers and IT operations staff to improve the software delivery time. At the heart of DevOps is Infrastructure as Code (IaC), an approach for automating infrastructure provisioning based on software development practices that emphasize using consistent and repeatable routines.

According to Achar (2020b), instead of racking servers and plugging in network cables, many sysadmins are applying the Infrastructure as Code approach to provide those resources using DevOps tools such as Ansible, Chef, Puppet, and Terraform. As a result, developers and ops teams are focusing their efforts on working on writing and executing code to define, deploy, and update the cloud infrastructure.

Enterprises are moving to the cloud to deal with privacy, security, performance, fault tolerance, and latency requirements. For this reason, enterprises are using multiple clouds to increase fault tolerance and allow for more graceful recovery from cloud provider outages. In this context, the industry has coined the term multi-cloud to refer to multiple clouds. However, multi-cloud deployments add an extra layer of complexity since each provider has its APIs, tools, and workflows.

Literature Overview

In modern-day DevOps practice, enterprise SaaS applications must employ a multi-cloud strategy to improve the application's performance, availability, and uptime and ultimately provide a better user experience (Pasupuleti, 2016). In addition, continuous management and monitoring are crucial to controlling day-to-day operations, such as recovering from errors and cloud bursting. A multi-cloud strategy would go a long way to handle these edge cases effectively. Still, the complexity of setting up the environment for multi-cloud is a significant pain point and barrier to adoption (Achar, 2015).

Some cloud providers have already implemented their infrastructure deployment services that automate the creation and management of cloud resources, such as Google Cloud Deployment Manager, AWS CloudFormation (Achar, 2020c), AWS OpsWorks, and Stacks (Mantoux, 1983). On the other hand, the DevOps community has also developed several open-source tools for managing the provisioning of the infrastructure of major cloud vendors, such as Heat (Frey, 2019) and Terraform (Pasupuleti & Adusumalli, 2018), and tools for installing software in existing servers. In previous work by Achar (2020a), empirical proof is provided of a multi-cloud infrastructure provisioning tool to compare with the code-centric Ansible deployment model. Adusumalli (2019) discuss a data-driven model for multi-cloud service management as code. Frey (2019) propose IaC as a best practice to speed up the DevOps lifecycle. Eric Wright also published a book on "DevOps Automation with Terraform and VMware" (Miah et al., 2021). Meanwhile, Jagdish Patni et al. have overcome the drawbacks associated with the manual infrastructure deployment and management approach by implementing a 3-tier architecture using Azure ARM templates, Pulumi, and Terraform.

Fundamental Techniques and Technical Differences

Multi-Cloud

The multi-cloud refers to using two or more public cloud platforms or services. With broader cloud adoption, application outcomes are essentially calling the shots: rather than standing up an IT environment to accommodate some unknown future peak capacity, modern architectures can dynamically deploy resources to suit application demands. As a result, more enterprises are finding that a multi-cloud environment that uses more than one public cloud service, performing different functions, offers the best of both worlds.

Now enterprises are recognizing the requirement of a multi-cloud SaaS offering, as it has several benefits such as lower risk, resilience, flexibility, cost-effectiveness, and performance efficiency, as well as avoiding vendor lock-in (Ruttan, 2006). Furthermore, by utilizing resources from multi-cloud providers, enterprises can consume the best services from each cloud vendor rather than sticking to an inferior service available from a single cloud (Fadziso et al., 2018). These public cloud providers could be anything from AWS, Azure, Alibaba, GCP, IBM, Oracle, Linode, Digital Ocean, which offer SaaS, IaaS, or PaaS offerings.

Infrastructure as Code (IaC)

Building infrastructure is an evolving process often requiring repetitive tweaks and improvements involving aspects such as scalability, performance, fault tolerance, and maintainability. Building and deploying infrastructure components in traditional environments was a manual and tedious task, translating to delays and decreased organizational agility. With the emergence of IaC, infrastructure components are now treated as merely software constructs—a code that can be shared across different teams. According to Sandobalin et al., Infrastructure as Code (IaC) is defined as an approach to infrastructure automation based on development and operations (DevOps) practices that promote continuous collaboration between developers and operations staff through a set of principles, techniques, and tools to optimize software delivery time.

To summarize this definition, an IaC solution complies with the following principles:

- Version control is a popular concept wherein every release corresponds to a source code build maintained as a versioned artifact in the environment. In IaC, a similar principle is applied to manage the infrastructure and changes using version-control commits in the source code repository. This provides traceability of changes made to the infrastructure definition, covering who made changes, what has changed, and so forth. This is crucial when rolling back to a previous code version while troubleshooting an issue.
- Predictability refers to IaC capability as a solution to always provide the same environment and associated attributes (as defined in the version-controlled system) every time it is invoked.
- Consistency ensures that multiple instances of the same baseline code provide a similar environment. This avoids inconsistencies and configuration drift when manually building complex infrastructure entities.
- Repeatability is a solution that always provides the same results based on the input.
- Composability refers to a service managed in a modular and abstract format, which can be used to build complex application systems. This feature empowers users to focus on the target application build rather than worry about the under-the-hood details and complex logic used for provisioning.

IaC Concepts

- Every infrastructure resource/component is declared as code, including packages, directories, user accounts, utilities, and configurations.
- The source code for each IaC tool is referred to by a specific term, e.g., cookbooks, recipes, playbooks, manifests, and templates.
- IaC introduces aspects of repeatability, significant speed improvements, and increased reliability.
- IaC offers consistency in the build. For example, if you need to manage several environments (e.g., development, QA, staging, and production), spinning those up from the same codebase ensures that negligible configuration drift is introduced across the environments, thereby maintaining the sanity of the domain.

Comparative Evaluation of Infrastructure Code

An infrastructure codebase includes many aspects, from specifying infrastructure resources to establishing distinct instances of other related resources to managing the supply of numerous interdependent framework pieces. As a result, various techniques for assessing different types of IaC.

Configuration Language

Most IaC tools support two primary language constructs: imperative and declarative language. Critical code refers to a procedural set of instructions that specify how to perform a task. Ansible and Chef work in a procedural style, which means they identify step-by-step codes and how to get to the desired end state. On the other hand, declarative code specifies the desired end state but not the method for completing it. For example, AWS CloudFormation, Google Deployment Manager, Terraform, SaltStack, Pulumi, and Heat all work in a declarative style of code. IaC encourages a declarative style of code wherein the desired end state and the configuration is present before the final form is provisioned. However, declarative code tends to be more reusable in the environment as current configuration changes are considered while catering to any new request for new infrastructure.

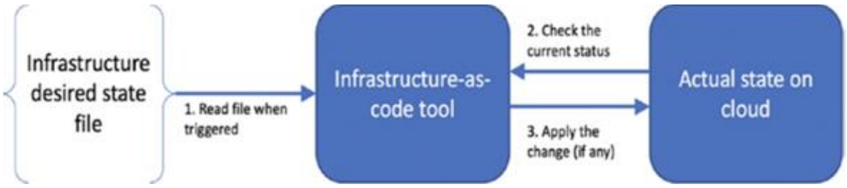
Infrastructure desired state file
1. Read file when triggered
Infrastructure-as-code tool
2. Check the current status
3. Apply the change (if any)
Actual state on cloud

Figure 1: A high-level view of how declarative code works

Configuration Management Tools Versus Orchestration Management Tools

Configuration management tools are software engineering tools that use a domain-specific language (DSL) to handle changes in software projects. For example, Ansible, Chef, and Puppet are tools commonly used to configure servers. Their languages give a framework for creating and modifying concepts such as packages, files, services, and user accounts.

Orchestration management tools use a DSL (Domain Specific Language) to manage stacks: AWS CloudFormation, Google Cloud Deployment Manager, Heat, and Terraform, among others, are examples of tools used to manage stacks. They expose concepts from their domain and infrastructure platforms, allowing users to easily create code that refers to virtual machines, disk volumes, and network routes. In addition, some orchestration management tools like Pulumi deliver the same infrastructure as code workflows using standard general-purpose languages such as JavaScript, TypeScript, Python, and Go.

Basic IaC Workflow

Figure 1 introduces the basic workflow for a declarative style IaC solution. The development and implementation of IaC are broken into write, plan, and apply stages.

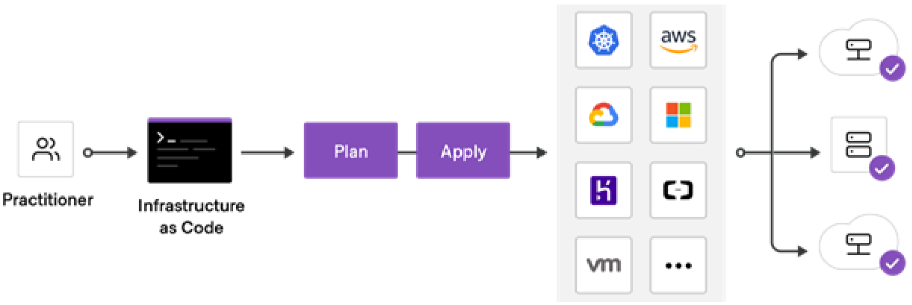

Figure 2. High-level workflow for IaC covering write, plan and apply stages

Write

During the writing stage, the primary focus is on the development of the necessary code to drive the plan. In this section, the infrastructure resources that are necessary for the providers and services that are required are defined. The code is contained in configuration files. For example, a user wishes to create a mini cluster of virtual machines in Amazon Web Service (AWS). Three virtual machines, a Virtual Private Cloud (VPC) network, this cluster needs both a security group and a load balancer service in order to function properly. The configuration files contain the code declaring each resource separately with corresponding parameters.

Plan

The next step, which comes after the writing of the configuration code, is to execute the IaC tool in order to generate a plan for the system. A local Command Line Interface (CLI), the Cloud Console, or other high-level languages that are able to interface with the framework via the Cloud Development Kit are used to run the device (CDK). When the plan is executed, a search of local directories for configuration files will be performed, and the results will be processed into a list of actions that will be sent to the provider (s). This list is the execution plan, and it includes all of the steps required to create, update, and delete the target infrastructure so that it matches what is declared in the configuration code. In addition, the generation of the plan is dependent on the preexisting infrastructure that is present in the state. The state provides specific information on all of the infrastructure resources that are currently available. Either locally in the file system or remotely in another location is where the state file can be found. For instance, if a configuration includes an Elastic Compute (EC2) virtual machine called "VirtualMachine1," the phrase "for example" would be appropriate. After carrying out the plan, the currently active state file will be examined. When the program detects that the VirtualMachine1 EC2 already exists, it will not create a new instance of it. Either an update action will take place, or there will be no operation.

Apply

The process of applying is the very last step in the IaC workflow. When the tool is used on a plan, each action is carried out against the provider that corresponds to it. The IaC tool engages in conversation with its provider plugin, which then makes a call to the Application Programming Interface (API) of the cloud service provider with which it is affiliated (e.g., AWS). The configuration code and the one-of-a-kind API that is defined by each infrastructure provider are separated by an abstraction that is provided by provider modules. As part of the apply process, the state is modified to accurately reflect any changes to the infrastructure being targeted.

AUTOMATING MULTI-CLOUD SERVICES USING IAC

Automating multi-cloud management processes, including application development, delivery, and operations, using enterprise-supported multi-cloud automation solutions is recommended.

Image Management Tool

Image management has been a fundamental prerequisite for virtual or physical system provisioning. Traditional image automation solutions leverage baselines or golden images that were manually built and maintained. However, human errors introduced at the image-build stage could lead to configuration drift in the provisioned service.

It's best to adopt a tool for creating gold images for multiple platforms from a single source configuration, thereby solving problems with manually created images. Such a tool lets you automate the build of golden images. Ideally, it should work with tools like Ansible to install software while creating illustrations. For example, it uses configuration files, builders' concepts, and provisioners to spin up and configure an instance as a golden image. The configuration code can be changed in case of introducing a new state element (addition of a new agent) or during updating scenarios (patching, hardening) of the golden image and is used to create an updated image without human intervention.

The following are the key advantages of an image management tool:

- Accelerated image creation and update process: The tool helps create and update images belonging to numerous clouds or OS types within minutes. As a result, you don't have to wait for the administrator to make/update manually, which can take hours or even days.
- Support for multiple providers: The tool supports multiple providers and platforms to manage identical images across your multi-cloud environment with the same standardization and consistency level.
- Reduction in human error–induced inconsistencies: Using a codified approach for managing images, you can remove any inconsistencies or configuration drifts in your environment.

A Multi-Cloud IaC (infrastructure as code) Tool

A multi-cloud IaC (infrastructure as code) tool allows users to define a desirable infrastructure in a declarative language. This can be a domain-specific (DSL) or a standard programming language like GO or Python. The orchestration process of a multi-cloud platform is very complex due to the interoperability issue between clouds. Vendor-specific IaC tools do not support multi-cloud deployments. Pulumi and terraform are examples of third-party IaC tools that support multiple vendor deployments. Using IaC tools, the infrastructure components within the multi-cloud environment can be deployed and treated as code that you can version, share and reuse.

To illustrate the concepts, the following program shows how to create a Terraform configuration that provisions a GCP compute engine spot instance:

```
resource "google_compute_instance" "spot_vm_instance"
{
  name         = "spot-instance-name"
  machine_type = "f1-micro"
  zone         = "us-central1-c"

  boot_disk {
    initialize_params {
      image = "debian-cloud/debian-11"
    }
  }
  scheduling {
    preemptible                 = true
    automatic_restart           = false
    provisioning_model          = "SPOT"
    instance_termination_action = "STOP"
  }

  network_interface {
    # A default network is created for all GCP
projects
    network = "default"
    access_config {
    }
  }
}
```

This can also be achieved using Pulumi (another vendor-agnostic IaC tool) using JavaScript code. The following example uses the @pulumi/GCP package to create and manage three Google Cloud resources: a GCP.compute.Network in which the virtual machine will run, a gcp.compute.firewall, which allows access for incoming SSH access, and a gcp. Compute. The instance is created inside the network from the Debian 11 base image.

```
const gcp = require("@pulumi/gcp");

// Create a network
const network = new gcp.compute.Network("network");
const computeFirewall = new
gcp.compute.Firewall("firewall", {
    network: network.id,
    allows: [{
        protocol: "tcp",
        ports: [ "22" ],
    }],
});

// Create a Virtual Machine Instance
const computeInstance = new
gcp.compute.Instance("instance", {
    machineType: "f1-micro",
    zone: "us-central1-c",
    bootDisk: { initializeParams: { image: "debian-
cloud/debian-11" } },
    networkInterfaces: [{
        network: network.id,
        // accessConfigs must include a single empty
config to request an ephemeral IP
```

```
        accessConfigs: [{}],
    }],
});

// Export the name and IP address of the Instances
exports.instanceName = computeInstance.name;
exports.instanceIP =
computeInstance.networkInterfaces.apply(ni =>
ni[0].accessConfigs[0].natIp);
```

The following are the key benefits of using an IaC tool to provision infrastructure:

- Accelerated multi-cloud service provisioning: A multi-cloud IaC tool enables accelerated provisioning of services across multiple clouds, covering a diverse range of technologies.
- State management:

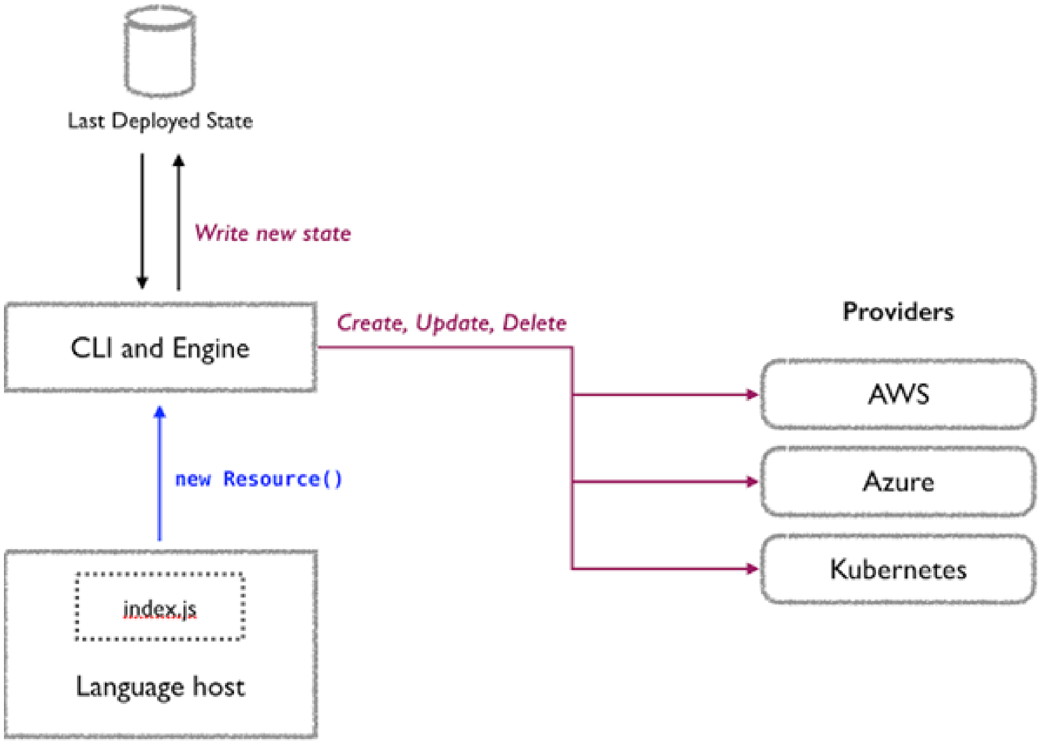

Figure 3. A high-level overview of state management in a Pulumi program

The tool allows tracking services for changes or configuration drifts. This enables service configuration governance beyond the service lifecycle's provisioning phase.

- Planning and testing services: It enables the planning and testing of services before the provisioning or modification stages, allowing users to manage the service lifecycle safely and predictably.

Secrets Manager

Protecting secrets and access for automation is of primary importance. Secrets are any form of sensitive credentials that need to be controlled; they are used to unlock sensitive information. For example, a secrets manager is leveraged for storing and securely accessing secrets via API keys and passwords.

Workload Manager

A workload manager enables users to schedule tasks and deploy applications in a containerized or non-containerized infrastructure. It allows you to write code and build software using declarative infrastructure as code.

Service Mesh

A multiple–data center service mesh solution provides the capability to govern application service communication using a control plane. It also offers service discovery and health checks. It leverages a secure TLS protocol to establish mutual TLS connections. A service mesh allows you to control communication between different application components or between multiple applications. A service mesh leverages the IaC concept to define a communication policy. It typically uses a network proxy or sidecar concept for governing communication between application services. Data communication patterns help developers optimize service interaction and performance. For example, a service mesh can monitor the time reconnecting to access the application service during unavailability. This can help developers redefine the waiting period before an application service tries to reconnect.

Machine Provisioning Tool

The fundamental challenge developers face the consistency of the development environment used for writing code. Multiple solutions are available on the market, including VirtualBox, VMware Workstation, and Docker. Hypervisor platforms like VMware, KVM, and Hyper-V are typically used for setting up developer workstations; however, manual administration makes it tedious to manage configuration requirements for each application team, resulting in no consistency between different environments and introducing configuration drift due to manual intervention. A machine provisioning tool enables you to build and manage a developer's environment using a workflow-driven approach that leverages the power of infrastructure as a code. Using its integrations with various platform technologies, the developer environment is configured using a consistent, repeatable, and accelerated approach. From a developer's perspective, all the required software, utilities, and environment configurations can be applied to the environment using a configuration file. In addition, it enables application team members to use the same standard platform for development, allowing the developers to focus on development using their favorite software and tools without worrying about the underlying platform.

Continuous Delivery Tool

As modern infrastructure becomes more complex with the rise of public cloud IaaS and PaaS services and container/microservice/serverless-based applications, it's difficult for developers to keep track of deployment approaches in every platform (VM-based configurations, YAML files, Kubectl, schedulers, etc.). It enables developers to define how an application is built, deployed, and released across platforms. It is not a package manager or replacement of solutions like Kubernetes. Instead, it enables the abstraction of build and deployment complexities using codified flow, which is versioned and controlled.

It leverages build packs to build applications for various languages and frameworks, which can be stored as artifacts. These artifacts can be deployed on multiple platforms, leveraging IaaS or PaaS services. In addition, with a Continuous Deployment solution, we can create a workflow to deploy application components that use other IaC automation solutions, such as an Image Manager (for defining baseline images), IaC tool (for illustrating desired state configuration), Secrets Manager (for managing secrets), Workload Manager (for application orchestration), or a Service Mesh (for managing Service to service connectivity).

Architecture and Design

Conceptual Solution

The infrastructure code tools considered in this thesis are the orchestration tools commonly used for orchestrating multi-cloud infrastructures. To compare the reusability of the code in the hybrid cloud environment, Terraform and Pulumi are analyzed based on some abstractions considered required for any given deployment. The comparative metrics shown between Pulumi and Terraform are the factors considered in this selection that is only a suitable approach for the implementation discussed in this thesis. Disclaimer: These metrics do not imply that Pulumi or other IaC tools cannot be an appropriate solution to achieve the targeted performance. This depends on user preferences and the technical know-how of the users regarding the chosen tool. The table below illustrates how valuable the metrics are when comparing Terraform and Pulumi as infrastructure orchestration tools concerning their support for multi-cloud infrastructure deployment.

Table 1: IaC tools metrics differences

Metrics	**Terraform**	**Pulumi**
Language Support	Domain-Specific Language example HCL	General Purpose Language example JavaScript
Conditional Statement	The conditional statement is straightforward (intermediate knowledge of programming)	The conditional information is accessible (advanced knowledge of programming)
Documentation	Adequate	Limited
Testing	Not suitable for testing	Suitable for testing
Community Support	Adequate	Inadequate
Advance Programming Knowledge	Not required	Required
Module	Present	Present
Supported Platforms	Multiple platforms	Limited platforms
Dependencies	Not required	Required
State Management	Local state by default	Pulumi cloud service by default

Using Pulumi to create a resource requires an advanced programming language, for instance, classes and functions, to combine the cloud providers' common abstractions. Community support for Pulumi is quite limited, which can cause delays in the implementation. In contrast, the functions of the Terraform module are equivalent to a traditional programming language. The modules in Terraform are reusable components. Terraform modules enable the grouping of codes into a logical cluster that can subsequently be controlled as a unit. It prescribes an argument by combining a subset of codes. Main.tf, variables.tf, and outputs.tf are the minimum deployment requirement. The main.tf file contains most of the functional code and variables.tf is used to store variables and outputs.tf is used to describe what is displayed at the end of a Terraform run. In this thesis, the choice for the multi-cloud providers is AWS and GCP.

Terraform Architecture

This section focuses in detail on the design and architecture of Terraform and Pulumi as they fulfill the requirements of tools that enable the secure build, change, and versioning of infrastructure codes. Their engines comprise a set of commonly used resource providers, including Azure, AWS, GCP, OCI, IBM, etc.

Terraform is entirely built on a plugin-based architecture. Terraform plugins enable all developers to extend Terraform usage by writing new plugins or compiling modified versions of existing plugins:

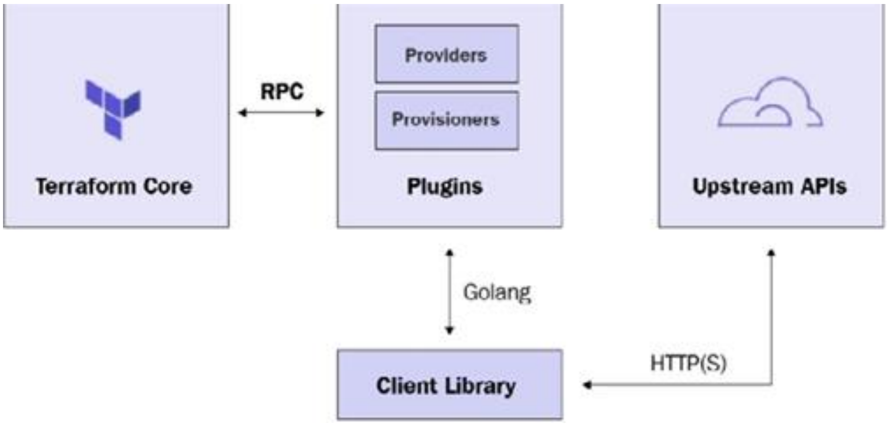

Figure 4: Terraform Architecture

As you can see in the preceding Terraform architecture, there are two key components on which Terraform's workings depend: Terraform Core and Terraform plugins. Terraform Core uses Remote Procedure Calls (RPCs) to communicate with Terraform plugins and offers multiple ways to discover and load plugins. For example, terraform plugins expose an implementation for a specific service, such as AWS, a provisioner, etc.

The responsibilities of Terraform Core are as follows:

- IaC: Reading and interpolating configuration files and modules
- Resource state management
- Resource graph construction
- Plan execution
- Communication with plugins via RPC

The responsibilities of provider plugins are as follows:

- Initialization of any included libraries used to make API calls
- Authentication with the infrastructure provider
- The definition of resources that map to specific services

The responsibilities of provisioner plugins are as follows:

- Executing commands or scripts on the designated resource following creation or destruction

To illustrate these concepts, the following Terraform script shows how to create a t2.micro-sized EC2 instance.

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 4.16"
    }
  }
  required_version = ">= 1.2.0"
}
provider "aws" {
  region  = "us-west-2"
}
resource "aws_instance" "web_server" {
  ami           = "ami-7879aa10"
  instance_type = "t2.micro"
}
```

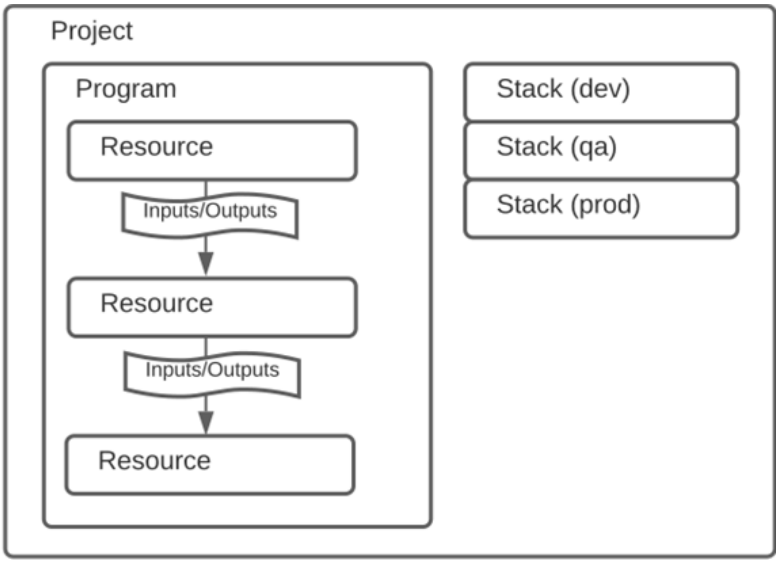

Figure 5: Pulumi Architecture

Written in general-purpose programming languages, Pulumi programs provide a description of how the components of your cloud infrastructure should be assembled. A project is a directory that stores programs. This directory contains the program's source code as well as metadata that describes how to run the program. You must first allot resource objects in your schedule that have properties that correspond to the desired state of your infrastructure before you can declare new infrastructure.

To illustrate these concepts, the following TypeScript program shows how to create an AWS Security Group named web-sg with a single ingress rule and a t2. micro-sized EC2 instance using that security group.

```
import * as pulumi from "@pulumi/pulumi";
import * as aws from "@pulumi/aws";

const group = new aws.ec2.SecurityGroup("web-sg", {
    description: "Enable HTTP access",
    ingress: [{ protocol: "tcp", fromPort: 80,
toPort: 80, cidrBlocks: ["0.0.0.0/0"] }],
});

const server = new aws.ec2.Instance("web-server", {
    ami: "ami-7879aa10",
    instanceType: "t2.micro",
    vpcSecurityGroupIds: [ group.name ], // reference
the security group resource above
});
export const publicIp = server.publicIp;
export const publicDns = server.publicDns;
```

Service Discovery Protocol

Terraform and Pulumi use a remote service discovery protocol (SDP) to implement most of their functionality. SDP is a protocol to advertise and discover network services and key/value storage. The service discovery tool is typically made up of a small number of nodes that constitute the consensus quorum and a more significant number of client nodes that run alongside your applications. The services are resolvable via DNS to an address with an HTTPS server on port 443.

Currently, terraform uses the following service identifiers:

- `Login.v1: login protocol version 1`
- `Modules.v1: modules registry API version 1`
- `Providers.V1: provider registry API version 1`

Dependency Mapping in Terraform and Pulumi

Basic Resources Mappings Terraform: Resources represent the fundamental units that make up your cloud infrastructure. For example, a resource block describes one or more infrastructure objects, such as virtual networks, compute instances, storage buckets, etc.

```
resource “type” “name” {
parameter = “foo”
parameter2 = “bar”
list = [“one”, “two”, “three”]
}
```

Where:

- **Resource** = Top-level keyword.
- **Type** = Type of resource.
- **Name** = A name refers to a single object that Terraform manages. The names of resources must always begin with the name of the provider that contains them, followed by an underscore. For example, a resource that is provided by the provider Flask could be named flask server.

For example:

```
resource "aws_instance" "flask_server" {
  ami                    = "ami-508d8g39"
  instance_type          = "f1.micro"
  monitoring             = true
  vpc_security_group_ids = [
      "sg-2547bcfg",
  ]
  tags          = {
    Name        = "Application Server"
    Environment = "production"
  }
  root_block_device {
    delete_on_termination = true
  }
}
```

Basic Resources Mappings in Pulumi: You can use Pulumi to provision cloud infrastructure using traditional programming languages such as Python, Go, JavaScript, TypeScript, Java, C#, and YAML. As an illustration, the import statement here imports the module object containing all the AWS deployment resources.

```
import * as aws from "@pulumi/aws";
import * as pulumi from "@pulumi/pulumi";
```

Every resource managed by Pulumi has a name you specify as an argument to its constructor. To provision a resource, the following syntax is used:

```
const foo = new aws.Thing("my-thing");
```

For instance, the logical name of this security group resource is flaskserver-secgrp:

```
const group = new aws.ec2.SecurityGroup("flaskserver-
secgrp", {
    ingress: [
        { protocol: "tcp", fromPort: 22, toPort: 22,
cidrBlocks: ["0.0.0.0/0"] },
    ],
});
```

The name that you designate during resource creation is used in two ways:

- To construct the Universal Resource Name (URN) used to track the resource across multiple updates. This is particularly useful when choosing between creating new resources and updating existing ones.

Data Flow Diagram

Pulumi Data Flow: Pulumi uses the desired state model to manage infrastructure. State stores metadata about your infrastructure; pulumi uses this information to determine when and how cloud resources should be created, read, deleted, or updated. A language host is used to compute and compare the desired state for a stack's infrastructure with the present state to decide which resources must be created, updated, or purged. The state is persisted in a backend, an API, and a storage endpoint of your choosing. Backend options include the default Pulumi Service application, an easy-to-use, managed application that takes care of the state and backend details for you. The web application can be accessed at app.pulumi.com or api.pulumi.com for the REST API. You can also use a local filesystem or cloud storage (Microsoft Azure Blob Storage, Google Cloud Storage, AWS S3, or any AWS S3 compatible server such as Minio or Ceph as your backend). Finally, the deployment engine uses a set of resource providers (such as AWS, Azure, Kubernetes, and so on).

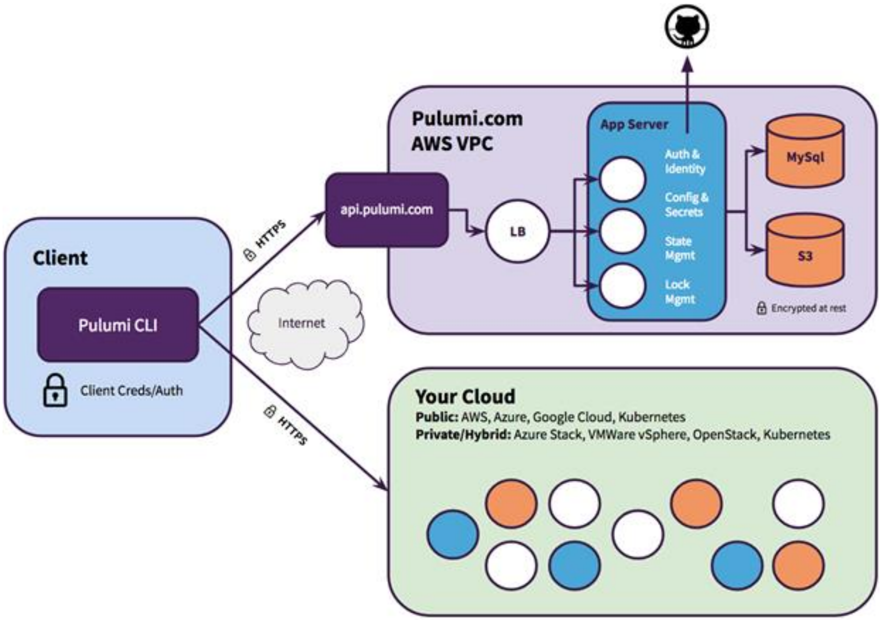

Figure 6: A Simplified Diagram of the Pulumi Data Flow

Terraform Data Flow: Terraform uses RabbitMQ, Redis, PostgreSQL, and Object Storage to store state. These coordinate services are stated in Terraform Enterprise—a paid, self-hosted distribution of Terraform for small teams. You can deploy Terraform Enterprise on the following:

- Amazon Web Services
- Google Cloud Platform
- Microsoft Azure
- VMware

You can also use the following:

- Terraform OSS (Free)
- Terraform Cloud (Paid - Saas Model)

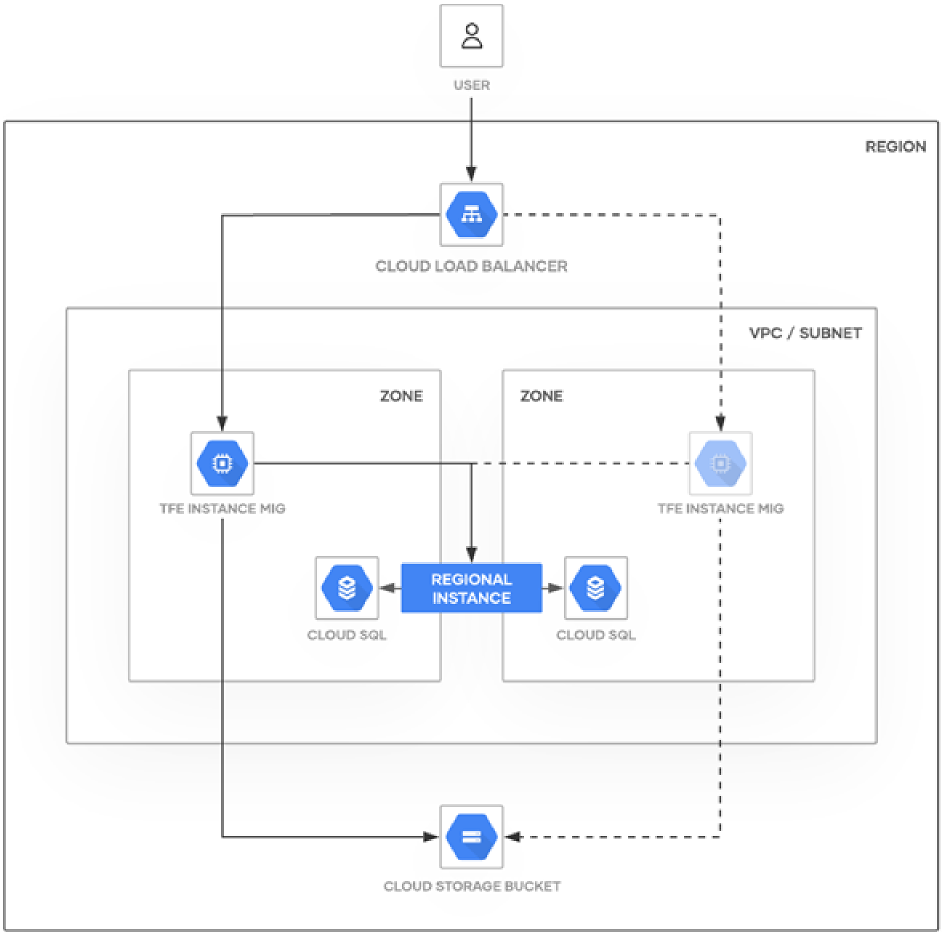

Figure 7: Data flow through the various Terraform Enterprise backend services deployed on GCP

All traffic is directed to the application layer, which is managed by a Regional Managed Instance Group. This group offers an auto-recovery mechanism in the event that an instance or Zone fails to function properly.

The Storage Layer is made up of several different service endpoints (Cloud SQL and Cloud Storage), and it takes advantage of the inherent resiliency that Google Cloud Platform offers. Through the use of the Cloud SQL endpoint, the Terraform application has established a connection to the PostgreSQL database. Likewise, all database requests are sent to the database instance through the use of the Cloud SQL endpoint.

Through the Cloud Storage endpoint that is associated with the bucket that has been defined, the Terraform application has a connection to object storage. Each and every request for object storage is forwarded along to the highly available infrastructure that underpins Cloud Storage.

Visualizing IaC Configuration

Terraform Graph generates a visual representation of the IaC configuration or execution plan. The output is a transparent rendering of the entire Graph, including the nodes that map onto natural objects in the design and the utility nodes that Terraform creates to handle the preparation.

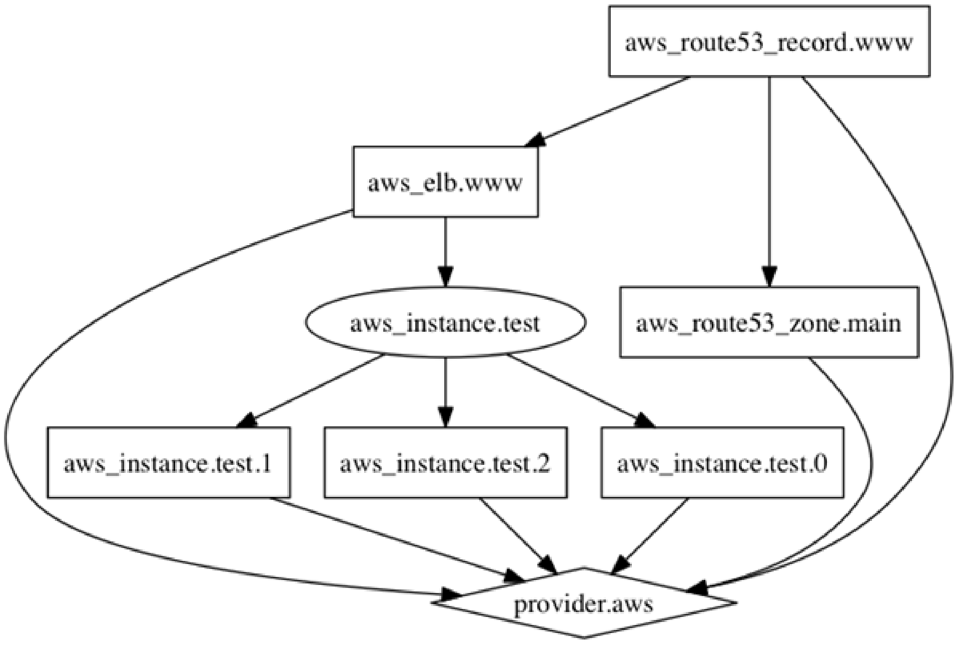

Figure 8: Hierarchical data structures Terraform on AWS using DOT format

The figure above shows the design architecture and variables employed to provision ELB, Route53 records, and EC2 instances on AWS with Terraform using the DOT layout. The DOT is a graph description Language "hierarchical" or layered drawings of directed graphs, an abstract grammar defining the DOT language. Each node in the Terraform graph performs some action on the plan or the state. The root structure is composed of the root provider (AWS).

Directory Structure of Multi-Cloud Infrastructure Using Tree

The standard module structure that Terraform recommends is a file and directory layout for reusable modules distributed in separate repositories. As illustrated below, the root module .terraform is the only required element for the standard module structure. This is where the Terraform files are stored. The directory structure presented below illustrates the system of Terraform modules consisting of AWS and GCP.

IMPLEMENTATION

In this section, we focus on the requirements for the implementation of this thesis. This includes some specific technologies to facilitate the interpretation of the implementation and a practical perspective of deploying object stores with similar resource parameters on both AWS and GCP using Pulumi and Terraform. In the following examples, we will provision object stores to store semi-structured data across different regions. These data stores will be secure, highly available, and fault tolerant.

AWS and GCP have their ways of defining these resources; hence, there are subtle differences in Terraform code to create infrastructure on each cloud provider. The combined resources from AWS and the GCP provider for object storage creation on a multi-cloud environment can be seen in the above table:

Table 2: Comparison of AWS and GCP Resources

Resource	AWS	GCP
Object Storage	Simple Storage Service (S3)	Google Cloud Storage (GCS)

Create Terraform Resources for AWS & GCP

To create two object stores on two different cloud providers using Terraform:

First create the aws.tf file:

```
provider "aws" {
  region = "us-west-1"
}
```

Create an S3 Bucket:

```
resource "aws_s3_bucket" "my-bucket" {
  bucket = aws_s3_bucket.b.id
}
```

Now gcp.tf

```
provider "google" {
project = "googleproject"
 region  = "us-west1"
 zone    = "us-west1-c"
}
```

Create a Google cloud storage (GCS) bucket:

```
resource "google_storage_bucket" "my-bucket" {
  location       = "US"
}
```

Provision AWS & GCP Resources Using Pulumi

```
import * as aws from "@pulumi/aws";
import * as gcp from "@pulumi/gcp";
```

```
// Create an AWS resource (S3 Bucket)
const awsBucket = new aws.s3.Bucket("my-bucket");

// Create a GCP resource (Storage Bucket)
const gcpBucket = new gcp.storage.Bucket("my-
bucket");

// Export the names of the buckets
export const bucketNames = [
    awsBucket.bucket,
    gcpBucket.name,
];
```

Conclusion and Discussion

This paper discussed the multi-cloud Infrastructure as Code (IaC) concept. The findings of the implementation in this thesis indicate that a reusable multi-cloud deployment is achievable using orchestration tools such as Terraform and Pulumi. These orchestration tools allow users to declare and provision infrastructure resources through code. This works through provider modules that translate the code into API calls specific to the provider. Not only is the infrastructure created programmatically, but it is also agnostic of the underlying provider (i.e., AWS and GCP). By utilizing this abstraction, vendor lock-in can be avoided, and portability between vendors can be increased.

References

Achar, S. (2015). Requirement of Cloud Analytics and Distributed Cloud Computing: An Initial Overview. International Journal of Reciprocal Symmetry and Physical Sciences, 2, 12–18. https://upright.pub/index.php/ijrsps/article/view/70

Achar, S. (2020a). Cloud and HPC Headway for Next-Generation Management of Projects and Technologies. Asian Business Review, 10(3), 187-192. https://doi.org/10.18034/abr.v10i3.637

Achar, S. (2020b). Influence of IoT Technology on Environmental Monitoring. Asia Pacific Journal of Energy and Environment, 7(2), 87-92. https://doi.org/10.18034/apjee.v7i2.649

Achar, S. (2020c). Maximizing the Potential of Artificial Intelligence to Perform Evaluations in Ungauged Washbowls. Engineering International, 8(2), 159-164. https://doi.org/10.18034/ei.v8i2.636

Adusumalli, H. P. (2019). Expansion of Machine Learning Employment in Engineering Learning: A Review of Selected Literature. *International Journal of Reciprocal Symmetry and Physical Sciences*, *6*, 15–19. Retrieved from https://upright.pub/index.php/ijrsps/article/view/65

Fadziso, T., Adusumalli, H. P., & Pasupuleti, M. B. (2018). Cloud of Things and Interworking IoT Platform: Strategy and Execution Overviews. *Asian Journal of Applied Science and*

Engineering, *7*, 85–92. Retrieved from https://upright.pub/index.php/ajase/article/view/63

Frey, C. B. (2019). The Technology Trap. Princeton University Press.

Mantoux, P. (1983). The Industrial Revolution in the Eighteenth Century. The University of Chicago Press. Chicago.

Miah, M. S., Pasupuleti, M. B., & Adusumalli, H. P. (2021). The Nexus between the Machine Learning Techniques and Software Project Estimation. *Global Disclosure of Economics and Business*, *10*(1), 37-44. https://doi.org/10.18034/gdeb.v10i1.627

Pasupuleti, M. B. (2016). Data Scientist Careers: Applied Orientation for the Beginners. *Global Disclosure of Economics and Business*, *5*(2), 125-132. https://doi.org/10.18034/gdeb.v5i2.617

Pasupuleti, M. B., & Adusumalli, H. P. (2018). Digital Transformation of the High-Technology Manufacturing: An Overview of Main Blockades. *American Journal of Trade and Policy*, *5*(3), 139-142. https://doi.org/10.18034/ajtp.v5i3.599

Ruttan, V. W. (2006). Is War Necessary for Economic Growth? Military Procurement and Technology Development. University of Minnesota, Department of Applied Economics, Staff Papers, 06-14. http://purl.umn.edu/13534

--0--