Choosing the Right IaC Tool for Building Reusable Cloud Infrastructure

Master of Science (Tech.) Thesis
University of Turku
Department of Computing
Software Engineering
2024
Niko Kalliomaa

UNIVERSITY OF TURKU
Department of Computing

Niko Kalliomaa: Choosing the Right IaC Tool for Building Reusable Cloud Infrastructure

Master of Science (Tech.) Thesis, 61 p.
Software Engineering
March 2024

This study examines the critical role of Infrastructure as a Code (IaC) tools in automating the provisioning, configuration, and management of cloud infrastructure, which is essential for the rapid growth and success of organizations in the cloud computing domain. By enabling cloud developers to define infrastructure in code format, IaC tools facilitate easy versioning, change tracking, and deployment automation. This capability ensures that infrastructure can be consistently and reliably deployed across various environments, significantly reducing the risk of errors and downtime. However, the implementation of IaC tools without adequate knowledge can introduce risks.

The research emphasizes the importance of selecting the right IaC tool, considering factors like scalability, security, and compliance enhancement. The study aims to compare the features, ease of use, scalability, security features, and integration capabilities of the most utilized IaC tools, providing insights into building reusable cloud infrastructure. Through a comprehensive exploration of cloud services evolution, prevalent cloud resources, and a detailed analysis of selected IaC tools, this thesis offers a comparative assessment of these tools across different cloud service providers. The findings, derived from empirical tests, offer a deep dive into the technical nuances, qualitative attributes, and integration potential of these tools, paving the way for future research in the field of cloud computing and infrastructure development.

Keywords: cloud computing, infrastructure as code, pulumi, terraform, ansible, devops

Table of Contents

List of Figures

Tables

1 Introduction

Choosing the right Infrastructure as Code (IaC) tool for cloud infrastructure is a decision that should be taken into consideration when starting to building cloud infrastructure and can have a significant impact on the success of your organization's cloud journey. With the rapid growth of cloud computing, IaC tools have become an essential tool for cloud engineers looking to automate the provisioning, configuration, and management of their infrastructure. [1]

IaC tools allow cloud developers to define their infrastructure in a code format, making it easy to version, track changes, and automate the deployment process. This makes it possible for organizations to build reusable infrastructure that can be easily deployed across different environments, such as development, test, and production. This helps to ensure that the infrastructure is consistent and reliable, reducing the risk of errors and downtime. [2] On the other hand, the IaC tools include also many risks when implemented without required knowledge in the field of cloud infrastructure. [3]

Furthermore, IaC tools can help organizations to save costs by automating the provisioning of resources, reducing the need for manual intervention, and making it easy to scale resources up or down as needed. Additionally, IaC tools can help organizations to improve security and compliance by providing a way to automate the provisioning of resources with necessary security and compliance settings. [2]

In this research, we try to point out the main differences between most used

IaC tools available at the present moment and what things should be taken into consideration when choosing the IaC tool for reusable cloud infrastructure. The research questions for this study are:

RQ1: How do the features and capabilities of different IaC tools compare when it comes to building reusable cloud infrastructure?

RQ2: How do ease of use and learning curve, scalability and performance, security and compliance features, and community support and documentation with different IaC tools compare?

RQ3: How does the integration with other tools and systems compare for different IaC tools when building reusable cloud infrastructure?

In the second chapter of this thesis, we start with a comprehensive exploration of the evolution and proliferation of cloud services. This analysis delves into the myriad advantages offered by cloud computing and enumerates the leading cloud service providers in the market, alongside a detailed examination of their diverse offerings. The third chapter shifts focus to a broad spectrum of prevalent cloud resources and technologies, including those specifically employed in our research.

Progressing to the fourth chapter, we introduce and dissect the concept of "Infrastructure as Code" (IaC), selecting a triad of IaC tools for an in-depth analysis within the scope of this thesis. This segment also entails a thorough investigation of the workflows and salient features encompassed by these IaC tools. Subsequently, in the fifth chapter, we apply the previously selected tools to architect tangible infrastructure across three disparate cloud service providers. This practical application facilitates an in-depth comparative assessment of the technical nuances, qualitative attributes, and integration potential of these tools, primarily grounded in the empirical tests conducted in this chapter.

The sixth chapter is devoted to an analytical review of the outcomes. This encompasses a dissection of the methodologies that led to these results, an interpre-

tation of the findings, and a contemplation of potential future research trajectories in this rapidly evolving domain.

Finally, in the seventh chapter, we synthesize the research into a cohesive conclusion. This chapter aims to encapsulate the essence of the findings and articulate a conclusive summary, reflecting on the insights gained and their implications in the broader context of cloud computing and infrastructure development.

2 Cloud Services

2.1 History and Development

It has been over fifty years after the first introduction about the first steps into cloud computing. Even though, the concept of cloud computing began to gain widespread adoption only after the early 2000s [4]. In this chapter, we will take a look at the history and development of cloud infrastructure, from its early beginnings to its current state.

The history of cloud computing can be traced back to early years of 1960s, when the concept of utility computing was introduced. Utility computing is a model where different kinds of computing resources (e.g. storage or processing power) are provided for user based on the demand, like other utility services like water or electricity. However, back in the 1960s, it was only concept, not reality, because the technology wasn't advanced enough. [5]

After the advent of virtualization technology, in the late 1990s and early 2000s the foundations for cloud computing started to form. Virtualization technology made it possible to create virtual machines that run on top of a physical infrastructure. This created a possibility to create virtualized environments which could be quite easily provisioned and managed. [5]

The first cloud services were launched by Amazon Web Services (AWS) in 2006. This included the Xen-baseds Elastic Compute Cloud (EC2) and Simple Storage

Service (S3). These were the first real instances where customers were able to rent computing resources on-demand. This was the beginning of new era in the field of cloud computing where customers were able to use the public cloud infrastructure provided by the cloud providers such as AWS, Google Cloud or Microsoft Azure. [4]

As the time passed, also the cloud infrastructure continued to evolve. New services and capabilities were added that provided a more powerful and versatile environment. The market began to expand making the new services available to customers. The services included two new concepts: platform as a service (PaaS) and software as a service (SaaS). This also led into development of different cloud models such as private and hybrid cloud models. [5]

Today, the usage of cloud infrastructure has become an essential part of many organizations' IT strategy. It provides a flexible, scalable, and cost-effective way to manage infrastructure, making it possible for organizations to respond quickly to changing business needs. Since the growth and development of cloud infrastructure has been significant for many years, it is likely that it will continue to play a vital role in the IT landscape for many years to come. [6]

As the cloud infrastructure continues to evolve, the ability to manage and operate it becomes more important, as well as the choice of the right tools to operate it. This is where the topic of choosing the right IaC tools for building reusable cloud infrastructure comes in, as it will enable organizations to effectively manage and operate their cloud infrastructure with less effort and more efficiency.

2.2 Benefits of Cloud Computing

Cloud computing has revolutionized the way organizations build, manage, and scale their IT infrastructure. While the digital landscape evolves, businesses are increasingly adopting cloud technologies to support innovation, improve efficiency, and achieve competitive advantages. In this chapter, we will explore the key benefits of

cloud computing that make it a necessary choice for organizations of all sizes [7].

2.2.1 Cost Savings and Scalability

One of the main reasons, why organizations are adopting cloud computing as part of their everyday work, is the cost-saving potential of cloud computing. Traditional on-premises infrastructure requires significant upfront investments in hardware, software, and maintenance, where cloud computing operates on a pay-as-you-go model which enables organizations to scale their resources based on the demand. The flexibility of cloud computing allows organizations to avoid over-provisioning and to pay only for the resources they are actually using, resulting in more cost-efficient operations. [7]

2.2.2 Enhanced Flexibility and Agility

Cloud computing offers unparalleled flexibilty and agility in deploying, managing, and adapting to changing business needs [7]. With IaC tools, such as Terraform, Pulumi, or Puppet, organizations can define and automate their cloud infrastructure. This makes it easier to provision and modify resources as required. By adopting this dynamic approach, organizations will get faster time-to-market for new products and product updates [8]. This empowers organizations to respond quickly to market demands and gives an advantage compared to their competitors [7].

2.2.3 Global Reach and Accessibility

The cloud surpasses geographical boundaries which allows organizations to access their applications and data from anywhere with an internet connection. The global reach does not only support the internal operations of an organization but also enables companies to reach their customers worldwide without the need for physical data centers around the world in different regions. [9]

2.2.4 Reliability and High Availability

Leading cloud service providers, such as Amazon Web Services, Microsoft Azure, and Google Cloud Platform, operate with redundant systems which ensures high availability and reliable access to resources. Additionally, Service Level Agreements (SLAs) guarantee the customer a certain level of uptime. This robust infrastructure and SLAs offer the customer an environment with no risk of downtime due to hardware failures and ensure continuous operations and optimal performance for critical applications. [7]

2.2.5 Improved Security and Compliance

Contrary to common misconceptions, cloud providers want to protect their customers' data and invest heavily in security measures. Cloud providers have and most likely will implement advanced security protocols, encryption mechanisms, and identity management to keep their customers safe from unauthorized access and other cyber threats [7]. Additionally, most of the major cloud providers comply with industry standards and regulations to help their customers to meet their data and privacy requirements, such as the General Data Protection Regulation (GDPR) in European Union [10].

2.2.6 Focus on Core Competencies

When companies offload the burden of managing hardware and infrastructure to cloud providers, they can focus more on their core competencies and strategic initiatives. The cloud allows teams to concentrate on developing innovative products, delivering exceptional customer experiences, and driving business growth, rather than using large amount of their time and resources to infrastructure maintenance.

2.2.7 Enviromentally Friendly

In the current state of the world, the environmental aspects should be taken into account in every business. Cloud computing contributes to a more sustainable future. Cloud data centers usually use advanced energy-efficient technologies and resource utilization. Organizations can play a part in promoting eco-friendly practises and mitigating environmental impacts by migrating to cloud. [7]

2.3 Cloud Providers

Even though, there is a myriad amount of cloud providers offering their cloud services on the market, three major companies cover about two thirds of the markets. In this thesis, we are mainly focusing on these three cloud vendors which are Amazon Web Services (32%, Q2 2023), Microsoft Azure (22%, Q2 2023), and Google Cloud Platform (11%, Q2 2023) [11].

2.3.1 Amazon Web Services

Amazon Web Services (AWS), a comprehensive cloud computing platform, was officially launched in 2006. AWS utilizes the XEN virtualization technology, a powerful and efficient open-source hypervisor that supports the creation and management of virtualized server environments. This technology underpins the AWS Elastic Compute Cloud (EC2), allowing users to deploy and manage virtual servers, known as instances. AWS offers support for seven different server operating systems including Amazon Linux, Cent OS, Debian, Oracle Linux, Red Hat Linux, Ubuntu, and Windows Server [12]. Table 2.1 presents an overview of the core services provided by Amazon Web Services.

2.3.2 Microsoft Azure

Microsoft Azure made its debut in 2010. As of the latest update, Azure supports 52 different regions, making it one of the largest cloud infrastructures in the world. At the heart of Azure's computing capabilities is Microsoft's proprietary virtualization technology, Hyper-V. This technology forms the foundation for Azure's virtual machines and services. Azure supports six different server operating systems including Cent OS, FreeBSD, OpenSUSE, Oracle Linux, Ubuntu, and Windows Server. [12] Table 2.2 presents an overview of the core services provided by Azure.

2.3.3 Google Cloud Platform

Google Cloud Platform (GCP) was launched in 2011. As of the latest information, GCP has expanded its global infrastructure to include 21 regions. GCP leverages the Kernel-based Virtual Machine (KVM) Hypervisor for its virtualization technology and supports five different Server Operating Systems including Cent OS, Debian, Ubuntu, Red Hat Linux, and Windows Server [12]. Table 2.3 presents an overview of the core services provided by Google Cloud Platform.

Compute services	
Amazon EC2	Virtual machines for scalable compute capacity
AWS Lambda	Serverless computing
Storage services	
Amazon S3	Object storage
Amazon EBS	Block storage
Amazon Glacier	Long-term archival storage
Database services	
Amazon RDS	Relational database management
Amazon DynamoDB DB	NoSQL Database service
Amazon Redshifts	Data Warehousing
Networking services	
Amazon VPC	Resource isolation in logically isolated network
Amazon Route 53	Domain name system web service
AWS Direct Connect	Dedicated network connections to AWS
Analytics and AI	
Amazon EMR	Large dataset processing and analyzing
Amazon SageMaker	Building, training, and deploying machine learning models
Management and Monitoring	
AWS CloudWatch	Resource and application monitoring
AWS CloudFormation	Template based IaC tool for AWS
Security and Identity	
AWS IAM	User account and permission management
Amazon GuardDuty	Threat detection and responding using machine learning

Table 2.1: AWS core services

Compute services	
Azure Virtual Machines	Scalable virtualized computing resources
Azure Functions	Serverless computing
Storage services	
Azure Blob Storage	Object storage
Azure Disk Storage	Block storage
Database services	
Azure SQL Database	Relational database service
Azure Cosmos DB	NoSQL database service
Azure Synapse Analytics	Big Data analytics service and Data Warehousing
Networking services	
Azure Virtual Network	Resource isolation to private network
Azure Load Balancer	Traffic distribution
Azure VPN Gateway	Secure connections between on-premises and Azure
Analytics and AI	
Azure HDInsight	Big data procession
Azure Machine	Machine learning model building, deployment, and management
Azure Cognitive Services	Pre-built AI models for vision, speech, language, etc.

Table 2.2: Azure core services

Compute services	
Google Compute Engine	Virtual machine instances
Google Kubernetes Engine	Container orchestration using Kubernetes
Google Cloud Functions	Serverless computing
Storage services	
Google Cloud Storage	Object storage
Google Cloud Persistent Disk	Block storage
Database services	
Google Cloud SQL	Relational database service
Google Cloud Bigtable	NoSQL database service
Networking services	
Google Virtual Private Cloud	Resource isolation to private network
Google Cloud Load Balancing	Traffic distribution
Analytics and AI	
BigQuery	Serverless data warehouse for analytics and business intelligence
Google AI Platform	Machine learning model building, training and deployment

Table 2.3: Google Cloud Platform core services

3 Cloud Resources

When building a cloud infrastructure, it is crucial to understand the key resources that form the backbone of applications and services. In this chapter we will go through the common cloud resources and their role in the modern digital applications.

3.1 Virtual Machines (VM)

Virtual machines play the main role in cloud computing. Cloud Providers offer several different virtual machines with different configurations and computing resources. The users are able to build their own virtualization environments with optimal amount of computing power, memory, and storage. Virtual machines provide a foundation for building, managing, and testing software across different environments. [13]

3.2 Storage

Cloud providers offer different types of cloud storage services for scalable and secure storage solutions. Cloud storages offer high data durability and availability by storing the data in multiple instances. [14]

Object storages, such as Amazon S3 or Google Cloud Storage, are built for unstructured data such as images, videos, and documents. Object storages allow

users to store and retrieve large amounts of this kind of data. Object-based storage systems are data storage architectures that treat data as distinct units, known as objects, in contrast to other contemporary distributed storage systems that manage data in blocks or files [15].

Block storages, such as Azure Disk Storage or Amazon EBS, are suitable for storing operating systems or applications (e.g. Docker images). Block storage refers to a method of data storage where files are divided into smaller segments, known as blocks, and are typically utilized in storage-area network (SAN) settings [15]. In block storage, each block functions like a separate hard drive, and it is set up by the storage administrator while being managed by the operating system of an external server.

3.3 Databases

Cloud databases, also known as DBaaS (Database as a service), are fully-managed database solutions. Even though databases can be hosted in virtualized cloud environment, users often prefer a out-of-the-box solution for databases to reduce the costs and complexity of managing the database. Cloud databases offer the user an optimized and scalable database instance with a simple setup process. [16]

3.4 Networking

Cloud networking enables the users to build private networks, load balancers, firewalls, and content delivery networks (CDNs) into the cloud instance. Virtual Private Clouds (VPCs) are isolated network environments in the cloud that allow secure networking between cloud resources. Load balancers distribute the incoming network activity evenly across multiple servers [17]. Firewalls are network security systems that monitor the incoming and outgoing network traffic based on user defined con-

figurations [18].

3.5 Containers and Orchestration

Containers, facilitated by technologies like Docker or Podman, provide a way of packing applications and their dependenciens into lightweight and efficient packages. These containers are able to share the kernel of the host machine which makes them more lightweight than traditional virtual machines. Kubernetes is a popular open-source container orchestration platform developed by Google. It allows automating, scaling, and managing the containerized applications. Building up a Kubernetes cluster requires an installation of master and worker nodes. The master node consists of four components: API server, scheduler, control manager, and ETCD storage system. The API server's responsibility is to handle the communication with the worker nodes. Scheduler and control manager manage the containers and positions them into the cluster. ETCD acts as a database for cluster information. It stores the data as key-value pairs. Containers and orchestrators simplify the deployment process, enchance portability, and ensure consistent efficiency across different environments. [19]

3.6 Serverless Computing

Serverless computing (e.g. AWS Lambda or Google Cloud Functions) enables the users to build and run applications without the burden of managing servers. In a serverless architecture, the resources are automatically scaled by the cloud provider based on the demand. This resource model can be very cost-effective, as the users only pay for the compute time that their functions consume. [20]

3.7 Identity and Access Management (IAM)

IAM is a service that provides a centralized control over the cloud resources' access and permissions. Adminstrators are able to define different roles and policies which ensures that non-authorized user's can not access the resources. IAM has the ability of simplifying user management in complex cloud environments. [21]

4 Infrastructure as Code

The concept of Infrastructure as Code has been around for a while. It's a methodology that treats infrastructure provisioning, configuration, and management as machine readable files instead of managing infrastructure manually through hardware or user interfaces. This empowers developers to manage their infrastructure with same tools and practises as they use for application code [3]. Changing from manual infrastructure management to automated processes brings numerous advantages.

Even though, the term "Infrastructure as Code" refers to all the tools that automate the processes for infrastructure management, in this thesis we are going to focus on tools that support the automation of provisioning, configuring and managing cloud resources.

At the present time, several IaC -tools have been introduced. In this thesis we, are going to select three IaC -tools for comparison to keep the scope of the thesis reasonable and still cover three different approaches for infrastructure challenges. The selection of IaC-tools for this thesis is mainly based on their popularity. Two other requirements are that the tool has to be an open-source project and support multiple cloud providers. Measuring the popularity of an IaC-tool can be a tedious task. We decided to compare the popularities of the tools by their Github stars since Github stars can be considered as proxy for popularity [22]. To select the best IaC-tools, we gathered a list of well-known IaC-tools and created a table that includes the name, a boolean value whether the tool is open source or not, a boolean value

whether the tool supports multiple cloud providers or not, and number of Github stars. Based on these criteria and the gathered data in our table 4.1, Terraform, Pulumi and Ansible were selected for this thesis.

Tool	**Open Source**	**Multi-Cloud**	**GitHub Stars**
Ansible	Yes	Yes	58,600
Terraform	Yes	Yes	38,800
Pulumi	Yes	Yes	17,600
Crossplane	Yes	Yes	7,500
Chef	Yes	Yes	7,300
Puppet	Yes	Yes	7,100
CFEngine	Yes	Yes	442
Google Cloud Deployment Manager	No	No	N/A
AWS CloudFormation	No	No	N/A

Table 4.1: Tool Comparison by capabilities and GitHub stars

4.1 Terraform

Terraform, Hashicorp's implementation of an IaC tool, was released in July 2014 [23]. The project was initiated by Mitchell Hashimoto and Armon Dadgar, co-founders of HashiCorp. Since then, it has grown into one of the most popular IaC tools, embraced by organizations of all sizes worldwide.

4.1.1 Key Features

Terraform offers the user a configuration tool that uses Hashicorp's domain specific language, HCL (Hashicorp Configuration language), or JSON to create declarative templates for provisioning a desired state of their cloud infrastructure [23].

Terraform has a provider model that allows the user to interact with APIs of various cloud providers, including Amazon Web Services, Google Cloud Platfrom, Azure etc [23]. This provider model acts as a bridge between Terraform and cloud providers' APIs.

Terraform uses a resource graph to create relationships and dependencies between separate resources [23]. This resource graph enables Terraform to determine the order in which the resources should be created or modified. Terraform's *graph* command can be used to generate a visual output of the resource graph. This output can be rendered using graph visualization tools like Graphviz, offering a tangible view of the infrastructure plan [24].

4.1.2 Terraform Workflow

Terraform deployment process starts with the implementation of Terraform configuration files (usually .tf files). In these files, the user defines all the required resources and their parameters.

After the implementation of configuration files, the next step is to download required plugins for chosen cloud providers. This can be done with a command:

```
terraform init
```

These plugins handle the communication between Terraform and cloud providers' APIs. [25]

When all the required plugins are installed, the user can proceed into the actual deployment phase. The terraform plan can be executed by command:

```
terraform apply
```

This command creates an execution plan defines the steps that are required to accomplish the required state of the infrastructure. In this phase, Terraform compares the current state of the infrastructure and the desired state that is defined in the configuration files and determines the required actions and the order of these actions to that are needed to match the defined infrastructure state. The planning phase also includes a step where the user is prompted with the actions that Terraform is going to apply. This step can be skipped with an argument *-auto-approve*. [26]

When the cloud resources are no longer needed, Terraform has an ability of deleting all the provisioned resources with command:

```
terraform destroy
```

This command destroys all the resources defined in the configuration files. This step can be considered as an opposite of the *terraform apply*. [27]

4.2 Pulumi

Pulumi is an open-source project founded in 2017 [28]. Pulumi enpowers developers to define, depoly, and manage cloud infrastructure using the most common programming languages including JavaScript (TypeScript), Python, Go, C#, VB, or Java. Users can also use Pulumi's own domain specific language (DSL), Pulumi YAML, if the use of programming language is not desired. Pulumi allows developers to use the same language for defining the infrastructure as they use for application code. This can lower the learning curve for developers and improves code the reusability of the code. Preview studies provide evidence of that Pulumi can decrease the time spent for developers to create deployments significantly [29].

4.2.1 Key Features

As previously highlighted, Pulumi distinguishes itself through its support for multiple programming languages, offering a significant advantage in terms of flexibility and functionality. This multilanguage capability is augmented by an integrated typechecking system, a feature that enhances the development process significantly. The typechecker is adept at identifying potential issues within the code during the development phase, such as invalid parameters or other anomalies, thereby preventing them from impacting the deployment stage. [29]

This preemptive issue detection is invaluable in ensuring that deployments proceed smoothly, minimizing the risk of runtime errors. Additionally, Pulumi enriches the user experience by incorporating built-in documentation for most supported languages. For instance, when utilizing TypeScript for Pulumi deployments, developers can access comprehensive class documentation directly through JSDoc. This feature eliminates the need to consult external documentation sources, streamlining the development process. [29]

The use of familiar programming languages for defining deployments also introduces the benefits of conventional coding practices into the realm of infrastructure management. Developers can effortlessly implement loops and conditional statements, making the code more intuitive and maintainable. Moreover, Pulumi allows for the incorporation of custom code blocks designed to validate and monitor the state of the infrastructure dynamically. An example of such functionality includes scanning for available IP addresses during deployment. This blend of programming language flexibility, integrated documentation, and advanced code functionality positions Pulumi as a robust and user-friendly tool in the landscape of Infrastructure as Code solutions. [29]

Pulumi offers a conversion tool that can transform deployment YAML-files into Pulumi code. This can ease out the process if the user wants to change their deploy-

ment from another tool to Pulumi. The tool supports transforming ARM, CloudFormation, Kubernetes CustomResource/YAML, or Terraform into Pulumi code. [30]

For deployment validation, Pulumi offers a Policy as Code tool, CrossGuard, which enforces the user to meet certain policies in their deployments. The policies can for example validate the certain resources have proper security parameters configured or block creation of public storage resources. Resource validations are written with JavaScript/Typescript or Python and different validations are created in a same way that the resources are created [31]. Also, there are emerging open-source libraries for Policy as Code that encourage the users to embrace the best practises for their Pulumi deployments [32].

To communicate with different cloud provider's APIs, Pulumi offers provider plugins that handle the communication with the API through Pulumi SDK. These modules are separate modules that encapsulate the logic that is needed to communicate with the provider's API. Along with official packages, Pulumi has several packages offered by the community. [33]

The Pulumi Registry serves as a hub for sharing and discovering reusable infrastructure components, empowering developers to leverage community-contributed packages and accelerate their development workflows. Additionally, Pulumi offers integrations with popular CI/CD tools, enabling seamless automation of deployment pipelines. [34]

4.2.2 Pulumi Programs

Pulumi deployment consist of *programs* which are written in general-purpose programming languages. These programs include the code where the user defines all the required resources as objects and defines how those should be composed. The basic structure of a Pulumi program is represented in Figure 4.1. Pulumi also offers

the user to define required dependencies between the resources.

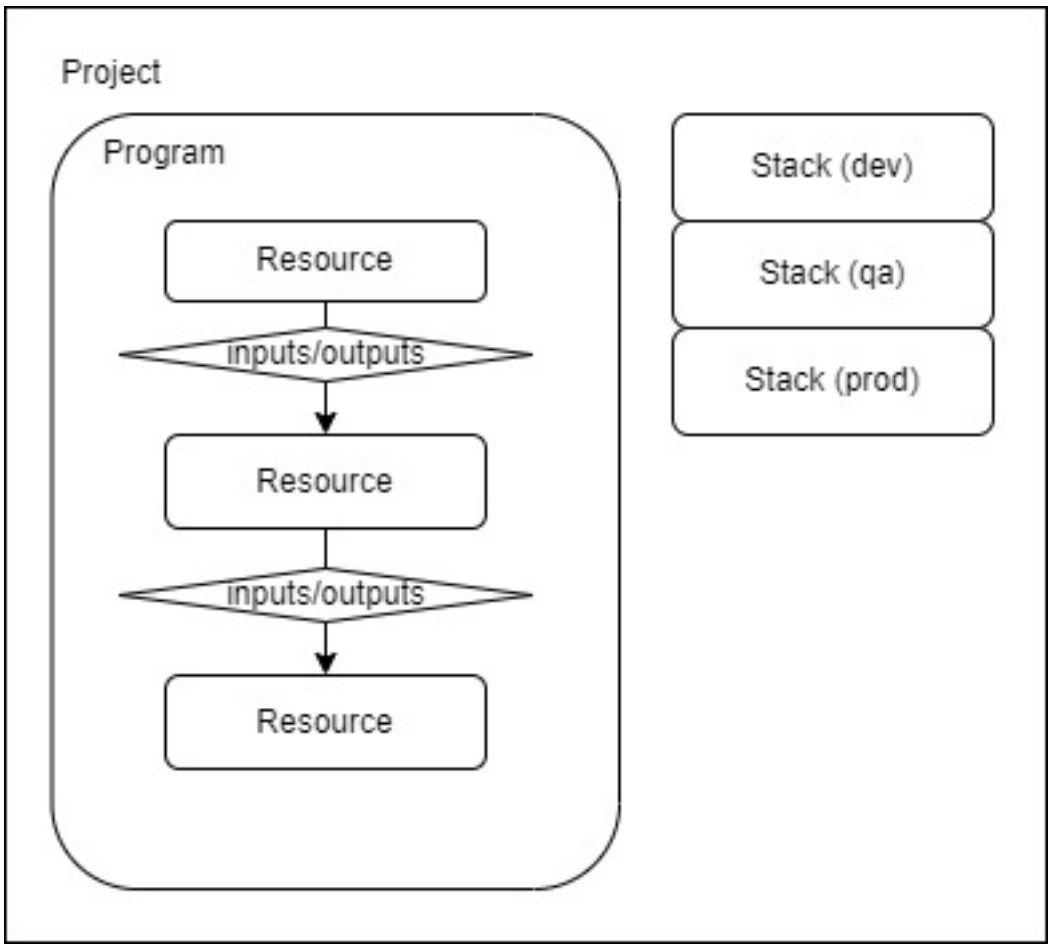

Figure 4.1: Diagram represents the basic components of a Pulumi program [33]

Along with programs, Pulumi uses *stacks* to store all the environment related information. Common use case for stacks is to use different stack for production and development releases. This enables infrastructure to be versioned and managed for different stages of the application lifecycle. Pulumi stack configuration is accessed via *Pulumi.Config* class [33]. Along with normal settings stored in stack's state, stacks may also include sensitive data, also known as *secrets*, that should be handled with care. For handling secrets Pulumi offers a out-of-box automatic encryption for stack values which are flagged to be secrets. If the user wishes so, they can use an alternative encryption provider such as AWS Key Management Service or Google

Cloud Key Management Service [35].

4.2.3 Pulumi Workflow

First step in creation a Pulumi program is to initialize the program. This can be done with the following command:

```
pulumi new
```

This script will go through required steps such as selecting the programming language and cloud provider. The script will download the required dependencies and initialize the program with necessary configuration files and boiler plate. [36]

After the project has been initialized, the user can define all the required cloud resources with their preferred programming language. Upon that, Pulumi employs the user with standard development practises such as version control, testing and code review. Pulumi's code-centric approach allows for rigorous testing, ensuring the reliability and correctness of the infrastructure code before deployment. Along with creating the program itself, the user needs to configure required stack variables.

The deployment process can be initialized with command

```
pulumi up
```

At this stage, Pulumi analyzes the code, creates a resource graph, and executes all required commands to align the infrastructure state with the state described in the pulumi program. Before actually applying the changes Pulumi prompts the user with review stage, ensuring control and transparency in the deployment process. This stage can be skipped with "-y" or "–yes" parameter. This is required when running the deployment process in CI/CD pipeline. [37]

For tearing down and deleting existing resources, Pulumi has a command

```
pulumi destroy
```

This command is opposite of *pulumi up*. Pulumi analyzes the current state of the infrastructure and the dependencies of different resources so that resources are removed in the correct order to prevent issues. This command also includes a preview prompt. [38]

4.3 Ansible

Ansible was originally released in 2012 by Michael DeHaan as a side project [39]. Unlike other tools, like Terraform and Pulumi, that are used in this thesis, Ansible is mainly designed to be a configuration management tool for existing cloud resources instead of orchestrating the whole infrastructure. However, Ansible is somehow capable of orchestrating the whole infrastructure. Ansible programs are built on *Ansible playbooks* which are YAML-files used for the infrastructure management. Ansible is written in Python and it can be extended with Python packages. Ansible playbooks can be considered to be idempotent, meaning that running the same playbook against same host will lead to same state in the mahcine. However, running custom scripts may lead to situation where Ansible does not know that a specific task had been already ran. This can be avoided by creating conditions for specific tasks that check if the machines state requires this specific task to be run. [39]

4.3.1 Ansible Workflow

As stated before, Ansible uses Playbooks as its configuration files. These playbooks are plain YAML-files. Before actually running Ansible, the user needs to have Python (2.6 or above) installed on their machine [39]. These Playbooks define the required tasks and the order of those tasks to accomplish the desired end state.

When the Playbooks are in place, the user needs to define a *Inventory file* that defines the machines Ansible should be run against [39]. These files can be written

in either JSON or INI format.

After configuring the Playbooks and a Invetory files,

```
ansible-playbook
```

command can be run to actually provision the desired actions. This command will create a SSH connection into the defined hosts in the Inventory file and run the tasks described in the Playbooks against these machines.

4.4 Reusability

In their thesis, Omofoyewa, Grebe, and Leusman [40] present a comprehensive analysis demonstrating the potential for reusability of Infrastructure as Code (IaC) tool templates in hybrid cloud environments. This research underscores the versatility of these templates, affirming their applicability across both private and public cloud infrastructures. Significantly, it elucidates how these templates serve as a foundational framework, enabling consistent deployment across various public cloud platforms. Furthermore, the study highlights the presence of numerous modular components within IaC tools. These modules encapsulate essential logical operations pertinent to the deployment phase, thereby facilitating their repurposing as reusable entities.

In addressing **RQ1:** "How do the features and capabilities of different IaC tools compare in the context of building reusable cloud infrastructure?" – and **RQ3:** – "How does the integration with other tools and systems compare for different IaC tools in the context of building reusable cloud infrastructure?" – it becomes imperative to conduct a detailed analysis. This analysis should focus on evaluating the extent to which the features and capabilities of these IaC tools bolster the reusability of templates. Such an evaluation is crucial for understanding the efficacy of these tools in facilitating a more streamlined, efficient, and flexible cloud infrastructure development process.

5 IaC -tool benchmarking

In the preceding chapter, we introduced three Infrastructure as Code (IaC) tools that form the analytical focus of this thesis. This chapter delves into practical applications of these tools, specifically examining their deployment in publishing applications and resources across three major cloud platforms: Google Cloud Platform (GCP), Microsoft Azure, and Amazon Web Services (AWS). Our objective is to deploy a web application on each of these platforms, employing all three IaC tools in the process.

The deployment strategy is structured into three distinct phases, each dedicated to adding a new layer or 'stack' to our application's architecture. This architecture follows the most common layer distribution in applications [41] where a frontend layer represents the user interface, backend represents the business logic layer, and a database layer represents the data access layer. These phases encompass the initialization of various infrastructure resources essential for launching the application. Key among these resources is the creation of a Kubernetes cluster, alongside a node pool within the cluster, which serves as the foundation for deploying containers and other services.

While it is possible to set up the infrastructure manually via the respective user interfaces of the cloud platforms, this thesis emphasizes the efficiency and effectiveness of using IaC tools for automation. By leveraging these tools, we aim to demonstrate a streamlined and replicable process for infrastructure deployment,

showcasing the capabilities, reusability, and comparative advantages of each tool within the context of cloud-based application publication. All the deployment files created during this research are available at [42].

5.1 Applications

The deployment strategy in this research encompasses a three-step process for launching an application in the cloud. This application is composed of three main components or 'stacks': a backend service, a frontend service, and a database. Unlike the backend and frontend services, the database will not be hosted within our Kubernetes cluster; instead, it will be situated in a managed database instance provided by the cloud provider.

In addition to these core elements, the deployment necessitates the establishment of foundational infrastructure resources, with the Kubernetes cluster being a central component. This cluster will serve as the operational environment for our backend and frontend services, offering a scalable, manageable, and resilient framework [19].

This strategic approach allows for the leveraging of cloud provider-managed services for the database, ensuring optimized performance, reliability, and maintenance [16], while the Kubernetes cluster efficiently handles the application services. By methodically building each layer, beginning with the base infrastructure and culminating with the user-facing frontend service, we aim to demonstrate a nuanced and comprehensive approach to cloud application deployment, highlighting the effective integration of managed services with custom-configured environments.

5.1.1 Database

As an initial step we will setup a database for our application. This step can be considered relative simple since we are utilizing the cloud providers' database

services.

Pulumi

As stated in chapter four, the Pulumi workflow starts by creating a new project. We initialize our project with the command **pulumi new**. This command prompts us with a question of which preconfigured stack we want to use. We will select the **<provider>-typescript**. After this step, Pulumi will ask us a few other variables for initialision of the project such as project name, project description, name of the initial stack and the default geographical location for the cloud resources. Pulumi will now initialize our project by installing the required npm packages and setupping configuration files which are **tsconfig.json** for the typescript configuration, **Pulumi.yaml** for the project configuration and **Pulumi.<stack-name>.yaml** for the stack configuration. More stack configuration files are created if user wants to setup multiple stacks.

After initializing the project, it's essential to define several key stack variables for optimal configuration. Key among these are:

1. **Region for Database**: This specifies the geographical location of our database.
2. **Database Variables**:
    - **Database Instance Version**: This determines the specific version of the database software we'll be utilizing.
    - **Database Tier**: This variable defines the performance and resource allocation tier of our database.
3. **Root Password**: A critical security measure, the root password for our database is marked as 'secure.' This designation ensures the password is stored in an encrypted format, bolstering our system's security. By default, we employ Pulumi Cloud's encryption method, but users have the option to integrate

their cloud provider's Key Management System for enhanced security measures (as suggested in [35]).

To set these stack variables, the command `pulumi config set <key> [value]` is used. For securing sensitive information like the root password, the `-secret` argument is appended, ensuring the variable is encrypted and stored securely.

When we have the stack variables in place, we will create the actual code to define the infrastructure. Config values can be extracted from a Config class provided by the pulumi npm package [43].

The last step in the provision process is to actually build the infrastructure. This process can be initialized with the command **pulumi up**. This command analyzes the current state of the infrastructure and creates the required actions to reach this state [37]. At this point, our infrastructure is empty so Pulumi will create the steps for setupping all the required resources. Before actually provisioning the infrastructure, the Pulumi CLI will prompt us with a list with the actions that are going to take place in the provisioning process which can be seen on the Figure 5.1. From this prompt, the user can either abort the provisioning or accept the provisioning plan and initialize the provisioning step.

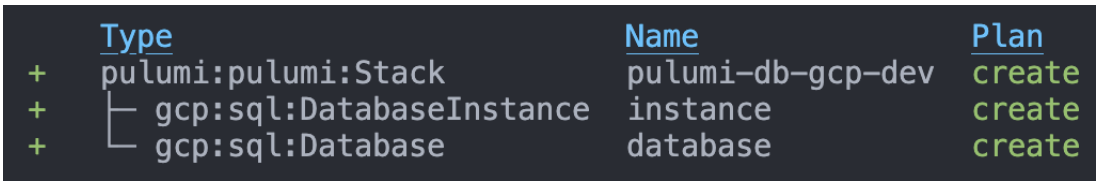

Figure 5.1: Pulumi prompt for creating a database instance into GCP

Terraform

Given that our current project focuses on deploying a single database instance, the configuration will be straightforward. We'll use a singular Terraform file, aptly

named *main.tf*, for this purpose.

In Terraform, the provider configuration is encapsulated within the *terraform* block. This block specifies the cloud providers required for the deployment, defined under the *required_providers* section. Additionally, we establish a specific *provider* block, where we define the essential elements for our chosen cloud provider, including:

- Credentials for Authentication
- Project Name
- Cloud Region

Once the provider configuration is set, we then define the cloud resources we intend to provision. For our project, these include:

1. **Database Instance**: The primary computing resource for hosting the database.
2. **Database**: The actual database for data storage and management.

Once we have all the configuration in place with desired parameters, we can deploy the resources into cloud. For this step we need to first run the **terraform init** command to tell the Terraform to download all the required packages for communicating with the cloud provider's API. After that is done, we can run the command **terraform apply** to initialize the deployment process. Terraform will first prompt us with the actions it is going to take and asks us the confirmation which we can see from Figure 5.2. When the confirmation is given, Terraform will create a database instance into the cloud provider's server.

Ansible

In the context of infrastructure provisioning, Ansible presents a notably simpler initialization phase compared to Terraform and Pulumi. The primary requirements for

```
Terraform will perform the following actions:

  # google_sql_database.database will be created
  + resource "google_sql_database" "database" {
      + charset         = (known after apply)
      + collation       = (known after apply)
      + deletion_policy = "DELETE"
      + id              = (known after apply)
      + instance        = "my-database-instance"
      + name            = "my-database"
      + project         = (known after apply)
      + self_link       = (known after apply)
    }

  # google_sql_database_instance.instance will be created
  + resource "google_sql_database_instance" "instance" {
      + available_maintenance_versions = (known after apply)
      + connection_name                = (known after apply)
      + database_version               = "MYSQL_8_0"
      + deletion_protection            = false
      + encryption_key_name            = (known after apply)
      + first_ip_address               = (known after apply)
      + id                             = (known after apply)
      + instance_type                  = (known after apply)
      + ip_address                     = (known after apply)
      + maintenance_version            = (known after apply)
      + master_instance_name           = (known after apply)
      + name                           = "my-database-instance"
      + private_ip_address             = (known after apply)
      + project                        = (known after apply)
      + public_ip_address              = (known after apply)
      + region                         = "europe-west6"
      + self_link                      = (known after apply)
      + server_ca_cert                 = (known after apply)
      + service_account_email_address  = (known after apply)

      + settings {
          + activation_policy     = "ALWAYS"
          + availability_type     = "ZONAL"
          + connector_enforcement = (known after apply)
          + disk_autoresize       = true
          + disk_autoresize_limit = 0
          + disk_size             = (known after apply)
          + disk_type             = "PD_SSD"
          + pricing_plan          = "PER_USE"
          + tier                  = "db-f1-micro"
          + user_labels           = (known after apply)
          + version               = (known after apply)
        }
    }

Plan: 2 to add, 0 to change, 0 to destroy.
```

Figure 5.2: Terraform prompt for creating a database instance into GCP

Ansible's setup are the installation of Python binaries necessary to operate Ansible and a credentials file to access cloud provider resources. Once the Ansible Python packages are installed, one can commence the development of an Ansible playbook for execution.

The first step in crafting this playbook involves defining essential variables needed for deployment, such as instance specifications, database name, and the path to the credentials file. This phase may also include the configuration of additional adjustable variables like the desired region or project name. These variables are all consolidated within the same playbook alongside other configurations.

The playbook will then outline two primary tasks for Ansible: creating a database instance and establishing the actual database. These tasks form the crux of the infrastructure provisioning process.

Executing the playbook is straightforward, accomplished by running the command **ansible-playbook <playbook-name>.yml**. During this phase, Ansible does not prompt for planning or confirmation but directly executes the tasks as defined. By default, the console logging is minimized, offering limited insight into the progression of the provisioning which is shown in Figure 5.3. However, users have the option to enhance the verbosity of the output, ranging from level 1 to 3, through parameters in the ansible-playbook command, allowing for more detailed feedback during the provisioning process.

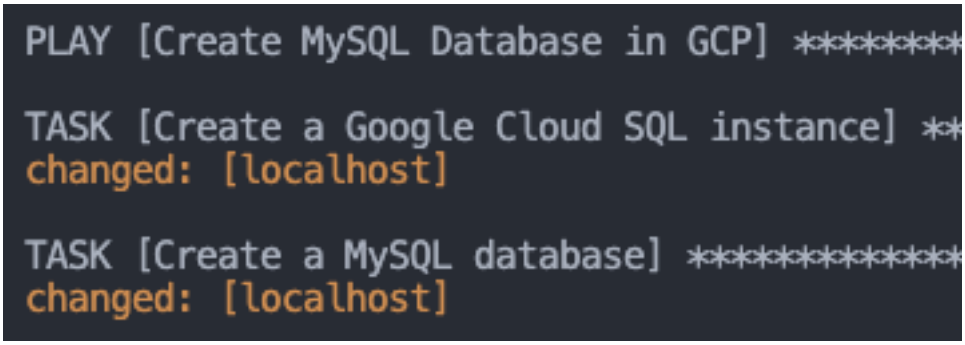

Figure 5.3: Provisioning progress visualisation in Ansible

5.1.2 Application Layer

Following the establishment of a database instance, the subsequent step involves constructing the application layer for the program, a process that is inherently more intricate due to the need for provisioning multiple resources. To achieve this, Kubernetes will be employed to establish a cluster, facilitating the deployment of both backend and frontend services. In this instance, the creation of a distinct networking layer over the applications will be omitted, primarily due to the absence of a

dedicated domain for the application. Instead, individual load balancer services will be utilized for each service, subsequently making these services accessible over the internet.

In preparation for the infrastructure provisioning, the Docker images for these services will be pre-emptively pushed into a container registry. This approach is designed to mirror a real-world scenario wherein these images are typically generated within a Continuous Integration/Continuous Deployment (CI/CD) pipeline and then transferred to a registry. This strategy ensures that the necessary components are readily available for deployment, streamlining the process of establishing the application layer.

Pulumi

The initialization phase of the application layer closely mirrors the process undertaken for the database setup, with a few additional nuances specific to application deployment. This phase begins with the creation of a new Pulumi program, a crucial step in laying the groundwork for the deployment. As part of this program, we must configure a series of stack variables, each tailored to meet the specific needs of the application layer.

Key among these variables is the designation of a namespace name, which serves as a unique identifier for this specific deployment within the Kubernetes environment. This namespace helps in organizing and isolating the resources for better management and security.

Additionally, we need to specify a name for our node pool within the Kubernetes cluster. The node pool is an essential component, as it comprises the worker machines (or nodes) that run our containerized applications. Proper naming and configuration of the node pool are vital for efficient resource allocation and management.

Another crucial step in this phase is the configuration of parameters for the services we intend to deploy. These parameters include:

The internal and external ports for the service: These ports define how the service communicates with other services within the cluster and how it is exposed to the outside world. The replica count: This determines the number of instances of the service that will run, impacting both the service's availability and load handling capacity. The URL of the service's image: This is where the Docker image for the service is stored in the container registry. Providing the correct URL ensures that the latest and appropriate version of the application is deployed. By meticulously defining these variables and parameters, we ensure that the application layer is not only correctly initialized but also primed for efficient and seamless operation within the Kubernetes cluster, contributing to the overall robustness and responsiveness of the deployed application.

After crafting the deployment files, we execute the **pulumi up** command once more, prompting Pulumi to efficiently provision the resources as defined in our configuration, seamlessly translating our code into the corresponding cloud infrastructure.

Terraform

The initialization process for deploying the application layer largely mirrors the steps undertaken during the database deployment phase. However, it necessitates specifying additional variables for the deployment. These include the locations of the Docker images, the configuration of internal and external service ports, the desired number of service replicas, and the specific Kubernetes namespace for our deployment.

Once these variables are defined, our next step involves outlining the essential resources. This encompasses the Kubernetes cluster, a tailored node pool for resource

allocation, the designated Kubernetes namespace, and the frontend and backend services. Each of these services will be coupled with load balancer instances to ensure efficient distribution of network traffic.

With the deployment templates prepared and verified, we proceed to execute again the **terraform apply** command. This action triggers Terraform to provision our defined infrastructure into the cloud environment, translating our configurations into a fully functioning cloud-based application layer.

Ansible

Like in the preceding phases, the configuration procedure with Ansible for the application layer exhibits remarkable similarity to the methodology employed for the database layer. However, the application layer necessitates some additional steps beyond those required for the database configuration. Primarily, it is essential to integrate the Kubernetes package. This integration is critical as it facilitates communication with the Kubernetes API, a vital component in the deployment process. Furthermore, the Ansible playbook requires augmentation through the inclusion of several pivotal variables. These variables include the names designated for the deployment and load balancer, as well as the URI corresponding to the docker image.

nce these preparatory steps are in place, the focus shifts to the deployment phase. It is noteworthy that the workflow within Ansible is characterized by its straightforward nature. The deployment process is initiated simply by executing the command: **ansible-playbook <playbook-name>.yml**. This command triggers Ansible to sequentially execute the predefined steps in the playbook.

Given the somewhat protracted nature of this process, it is advisable to employ the "-vv" parameter in conjunction with the aforementioned command. The utilization of this parameter is instrumental in generating a more verbose output. Such detailed output is beneficial as it provides clearer insights into the current stage of

the deployment process, thereby enhancing the understanding and management of the workflow.

5.2 Technical Comparison

Tool	**Pulumi**	**Terraform**	**Ansible**
Language	Programming language	HCL	YAML
State	Stored in cloud	Stored locally	Stateless
Execution model	Declarative	Declarative	Imperative
Resource visualisation	No	Yes	No
Task ordering	Yes	Yes	No
Destroy command	Yes	Yes	No

Table 5.1: Technical tool Comparison

Upon deploying a relatively straightforward application across three distinct cloud platforms utilizing three different Infrastructure as Code (IaC) tools, a comparative analysis reveals both similarities and divergences among these tools. In this section, we will analyse the technical differences between these tools to address **RQ1:** How do the features and capabilities of different IaC tools compare when it comes to building reusable cloud infrastructure?

5.2.1 State Management

Maintaining the state within the tool itself generally aids in aligning the tool with the actual state of the infrastructure under typical conditions. However, this approach can lead to discrepancies between the tool's recorded state and the real-time state of the infrastructure in the event of unforeseen circumstances. The distinction between

a stateless tool and one that internally manages state lies in the user experience; stateless tools, though demanding more effort from the user, offer a more adaptable environment for infrastructure management.

For instance, in a scenario where a user wishes to implement minor modifications to their infrastructure via the cloud provider's user interface, such an action is feasible with Ansible. However, with Terraform and Pulumi, changes must be exclusively made through the tool itself. Alterations made outside of these tools can result in a misalignment, as the tools may no longer possess an accurate understanding of the infrastructure's current state.

5.2.2 Tool Capabilities

In this analysis of infrastructure management tools, Terraform and Pulumi are observed to possess a broader range of inherent functionalities than Ansible, such as built-in infrastructure destruction, internal state and more comprehensive variable handling. Notably, Ansible necessitates user-generated destruction tasks, underscoring its primary design orientation towards the management of pre-existing infrastructure systems rather than the comprehensive provisioning, administration, and termination of such systems. Conversely, Ansible's installation process exhibits a high degree of simplicity, involving only the installation of Python binaries related to Ansible and the creation of executable playbooks. In contrast, Pulumi demands a more elaborate, multi-step installation process accompanied by extensive configuration. Terraform occupies an intermediate position in this spectrum, necessitating some degree of configuration, yet markedly less than that required by Pulumi.

Pulumi sets itself apart from other infrastructure management tools like Terraform and Ansible through its unique feature of enabling users to author Infrastructure as Code using mainstream programming languages instead of a domain-specific language. This capability has been linked with notable enhancements in

productivity among enterprises that have integrated Pulumi into their operations [44] [29] [45]. The utilization of a general-purpose programming language in an IaC context empowers users to execute a broader range of tasks beyond mere infrastructure provisioning. These tasks include scanning for available ports, generating deployment reports in file formats, and sourcing necessary data for deployment from external entities. In contrast, users of Terraform and Ansible are typically required to employ additional tools to manage these ancillary activities.

5.2.3 Task Ordering

Terraform uses a declarative approach, where the user specifies the desired end state of the infrastructure but not the steps to achieve it. Terraform automatically generates a plan based on the defined resources and their interdependencies. Its task ordering is managed through a directed acyclic graph (DAG), which Terraform constructs to understand the dependencies between resources [46]. This approach allows for parallel execution of non-dependent tasks, potentially speeding up the deployment process. However, it can lead to complexities when dealing with intricate dependencies or external changes to the infrastructure not reflected in the Terraform state.

Pulumi, in contrast, relies on imperative programming concepts, using standard programming languages like TypeScript, Python, and Go. Task ordering in Pulumi is explicit and controlled by the programming language's flow control mechanisms. For instance, you can use asynchronous functions or promises in JavaScript to manage dependencies. This approach provides a high degree of flexibility and can be more intuitive for developers familiar with these languages. However, it requires a deeper understanding of programming concepts and could lead to more complex code management challenges, especially in large-scale deployments.

Ansible, primarily a configuration management tool, executes tasks in the order

they are defined in its playbooks. These playbooks, written in YAML, describe a sequence of tasks to be executed on target machines. Ansible's approach is more linear and procedural compared to Terraform and Pulumi. While this can make the execution flow easier to understand and follow, it may lack the efficiency of parallel task execution seen in Terraform. Ansible is well-suited for scenarios where tasks need to be executed in a specific sequence, but it might be less efficient in handling large-scale or complex deployments with numerous interdependent resources.

5.2.4 Resource Visualisation

Terraform, with its declarative model, excels in resource visualization. It uses a directed acyclic graph (DAG) to represent the relationships and dependencies between resources. This graph can be visualized using the **terraform graph** command, which produces a visual output that can be rendered with tools like Graphviz. This visualization offers a clear, high-level view of the infrastructure layout, including dependencies, making it easier to understand and debug complex configurations.

Pulumi, while employing standard programming languages for infrastructure as code, does not inherently provide a built-in tool for visualizing resource dependencies in the same way as Terraform. Instead, developers can utilize the debugging and visualization tools available in their chosen programming language. This approach requires a more hands-on method for creating resource visualizations and might lack the immediate, out-of-the-box graphical representation of the entire infrastructure that Terraform offers.

Ansible, primarily focused on configuration management, does not have a native resource visualization tool akin to Terraform's graph. Being a procedural tool, Ansible executes tasks in the order they are defined in playbooks. While there are third-party tools and roles available that can help visualize the execution of these playbooks, they do not typically provide a holistic view of the entire infrastructure's

state or its interdependencies. Instead, these visualizations tend to be more focused on the sequence and results of task executions within a given playbook.

5.2.5 Resource Destruction

Terraform provides a straightforward and explicit command for resource destruction: terraform destroy. This command reads the current Terraform state and configuration files to determine which resources are managed by Terraform and systematically deletes them, following the dependencies defined in the resource graph. This ensures that resources are destroyed in a safe and orderly manner, preventing potential dependencies issues.

Pulumi handles resource destruction similarly, leveraging the programming language's syntax and structure. The process generally involves modifying the code to remove the resource definitions and then running pulumi up, which updates the infrastructure to match the current state of the code. Alternatively, Pulumi also offers a direct command, pulumi destroy, which removes all resources managed by a Pulumi stack.

Ansible, being predominantly a configuration management tool, does not have a built-in mechanism for destroying resources akin to Terraform or Pulumi. Instead, resource destruction must be explicitly defined in playbooks. Users write tasks to reverse the effects of what was deployed or configured, effectively undoing the actions previously performed. This approach requires a more manual and carefully constructed process to ensure that resources are removed correctly and safely.

5.2.6 Variable Handling

Terraform employs a strong, type-specific variable handling system. Variables are declared in Terraform configurations and can be assigned values from multiple sources, such as default values in the declaration, command-line flags, environment

variables, and separate tfvars files. Terraform's approach allows for flexible, yet structured variable management, supporting different types like strings, lists, and maps.

Pulumi, leveraging standard programming languages, handles variables just like any other programming variable. This means that variables in Pulumi are subject to the rules and capabilities of the language in use (like TypeScript, Python, or Go). This approach provides a familiar environment for developers and allows for complex data structures, conditional assignments, and computed values.

Ansible, on the other hand, uses a more straightforward and procedural approach. Variables in Ansible can be defined within playbooks, included in separate variable files, or passed at runtime. Ansible variables are often used to store values like server addresses, configuration parameters, and information about the environment. The tool provides flexibility in variable usage, supporting simple key-value pairs and more complex data structures like dictionaries and lists.

5.2.7 Results

In answering **RQ1**, this analysis elucidates that the optimal selection of an IaC tool hinges on the specific demands of the project, the expertise of the team, and the desired equilibrium between operational control and ease of use. Terraform and Pulumi emerge as particularly suited for comprehensive infrastructure management involving complex dependencies, with Terraform providing a user-friendly declarative syntax and Pulumi offering extensive flexibility through imperative programming. Ansible, with its focus on configuration management and procedural execution, is well-suited for scenarios where the infrastructure is pre-existing or less complex.

This comparative exploration underscores the imperative of aligning the chosen IaC tool with the project's unique requirements and goals, leveraging the distinct strengths of Terraform, Pulumi, and Ansible to foster the development of efficient,

scalable, and reusable cloud infrastructure solutions. In projects, where the DevOps team consists of software developers, Pulumi's utilization of programming tools can probably be utilized more efficiently, while the projects with separate DevOps team might consider this feature more like a burden.

5.3 Qualitative Comparison

In this section, we commence a detailed exploration of the non-technical distinctions among Pulumi, Terraform, and Ansible. We will employ a scoring system, ranging from one to three for each criterion, where a score of one denotes the best performance and three indicates the least favorable outcome. This evaluation primarily hinges on an analytical review of the technical disparities, supplemented by insights gleaned from the deployment exercises conducted earlier in this chapter. To attain a more nuanced and profound comprehension of these aspects, further extensive research would be beneficial. Such research could encompass a variety of methodologies, including comprehensive user interviews to capture firsthand experiences, in-depth analysis of official documentation for clarity and user support, and empirical studies focused on real-world application scenarios. This multifaceted approach promises to yield a more holistic and accurate assessment of the non-technical facets of these tools, offering valuable perspectives for users in choosing the right tool for their specific needs. Anyhow, in this section, we are going to find the answer to **RQ2:** How do the ease of use and learning curve, scalability and performance, security and compliance features, and community support and documentation with different IaC tools compare? This analysis is grounded on the insights gathered in the preceding sections.

5.3.1 Learning Curve

Terraform's learning curve is moderate. It requires understanding of HashiCorp Configuration Language (HCL), which is unique to Terraform but relatively straightforward. The tool's declarative nature simplifies the understanding of infrastructure state. However, grasping concepts like state management and resource dependencies can be challenging for beginners. Terraform's extensive documentation and large community support ease the learning process, but the depth of cloud concepts involved may require additional learning.

Pulumi presents the steepest learning curve, primarily due to its reliance on familiar programming languages like TypeScript, Python, or Go. While this is an advantage for those already proficient in these languages, it adds complexity for those who aren't. Understanding how traditional programming concepts translate to infrastructure provisioning can be challenging. The quality of Pulumi's documentation and community support is good, but the requirement for programming knowledge makes it more challenging for newcomers to IaC.

Ansible can be generally considered to have the lowest learning curve. Its use of YAML for playbooks makes it more accessible, especially for those with a basic background in scripting and configuration management. The procedural nature of Ansible, without the need for understanding complex state management or programming concepts, makes it easier for beginners. Ansible's strong documentation and large community further lower the barriers to entry.

In evaluating the learning curve for users with no prior background knowledge, we assign Ansible the highest rating for ease of learning, acknowledging its user-friendly approach and straightforward syntax. Terraform, with its balance of structured configuration and comprehensive documentation, secures the second position, offering a manageable learning experience. Pulumi, while powerful in its capabilities, is rated as having the steepest learning curve due to its reliance on familiarity with pro-

gramming languages and more complex infrastructure concepts. This assessment underscores the varying degrees of user accessibility and the initial effort required to master these tools.

5.3.2 Scalability and Performance

Terraform is renowned for its robust scalability and performance, particularly in managing large and complex infrastructures. Its declarative approach and efficient resource graph enable it to effectively handle dependencies and parallelize operations, thus optimizing deployment times and resource utilization. However, the state management in Terraform can become a bottleneck in extremely large deployments, which is an area for further investigation.

Pulumi offers considerable scalability and performance, especially since it leverages mainstream programming languages, allowing for sophisticated control over deployments. The imperative nature of Pulumi provides flexibility, but this can also lead to more complex scenarios in large-scale deployments, potentially affecting performance. The tool's performance in very large and dynamic environments is an aspect that warrants deeper exploration.

While Ansible is highly efficient in configuration management and automation, its scalability and performance in managing large-scale infrastructure deployments are somewhat limited compared to Terraform and Pulumi. Its procedural approach, relying on sequentially executing tasks, can slow down operations as the scale increases. Ansible's performance in massive, complex environments, and potential optimizations, remains an area for detailed study.

In our assessment of scalability and performance across these tools, Terraform emerges as the front-runner, earning the top score for its robust ability to efficiently manage large-scale and complex infrastructures. Its optimized resource graph and parallel execution capabilities significantly contribute to its superior performance.

Pulumi, with its flexibility and utilization of mainstream programming languages, secures the second place. While it offers substantial scalability, it faces challenges in handling extremely large and dynamic environments as efficiently as Terraform. Ansible, primarily focused on configuration management and automation, is assigned the lowest score in this category. Despite its effectiveness in smaller-scale deployments, Ansible's sequential task execution approach tends to limit its scalability and performance in comparison to Terraform and Pulumi, particularly in more extensive and complex infrastructure scenarios. This ranking underscores the varying degrees of efficiency and scalability inherent to each tool, offering a critical perspective for users considering the right tool for large-scale deployments.

5.3.3 Security and Compliance Features

Terraform offers robust security features, including state-level encryption and integration with HashiCorp Vault for secrets management. Its compliance capabilities are enhanced through custom policies and modules that enforce security best practices. Terraform's approach is centralized and declarative, which simplifies compliance audits.

Pulumi's security model leverages the inherent security features of the chosen programming language. It offers native secrets management and integrates with existing identity providers. Pulumi's approach to compliance is unique, as it allows the use of familiar programming constructs to enforce security policies.

Ansible's security focuses on secure SSH communication for configuration management. It provides role-based access control and integrates with various encryption solutions for secrets management. For compliance, Ansible offers modules that can be used to enforce policies and generate reports for audit purposes.

Terraform, with its integrated state encryption and seamless HashiCorp Vault integration, earns a score of 1 for its comprehensive approach to security and com-

pliance, especially beneficial in audit trails. Pulumi, utilizing the security features of its programming languages and offering native secrets management, receives a score of 2. Its approach is flexible but might require deeper programming knowledge for optimal security configuration. Ansible, focusing on secure SSH communications and modular encryption for secrets management, is rated 3. While effective in its domain, Ansible's security features are more decentralized, potentially necessitating additional configurations for compliance.

5.3.4 Community Support and Documentation

To analyze community support comprehensively, we employ a two-pronged approach. Firstly, we examine GitHub statistics to gauge the level of active contributions to the repositories of each IaC tool over the past year. This provides insight into the frequency and extent of community engagement in tool development and enhancement.

I. Moutidis and H. T. Williams [47] evidence in their research that Stack Overflow activity correlates with the popularity of a topic in real world. Based on this information, we utilize a specialized query to fetch data from Stack Overflow, focusing on questions that include specific keywords, namely the names of the IaC tools, within the past year. This query is pivotal in understanding the community's inquiries, challenges, and discussions surrounding each tool. Our Stack Overflow query is structured as follows:

```
select count(*) from Posts
where LOWER(Body) like '%<IaC-tool-name>%'
and ParentId is null
and CreationDate > DATEADD(year,-1,GETDATE())
```

This query is designed to count posts that specifically mention the IaC tool in question, providing a quantitative measure of the tool's presence and relevance

in community discussions. By combining these two methods, we aim to derive a holistic view of the community support landscape for each IaC tool, reflecting both developer contributions and user engagement.

Tool	Stack Overflow questions	Github contributors	Github commits
Pulumi	141	49	1481
Terraform	3801	19	868
Ansible	1826	18	676

Table 5.2: Tool Community Comparison

To gain a more nuanced understanding of these scores, further investigation is necessary. This could include conducting user interviews to gather firsthand experiences, analyzing user-generated content and discussions in community forums, and assessing the frequency and quality of documentation updates. Such detailed research would provide deeper insights into the actual effectiveness and user-friendliness of the community support for each tool.

Prior to the initiation of the deployment processes for the research conducted in this thesis, a thorough review of the "Get Started" sections provided by each tool was undertaken. Both Pulumi and Terraform offer comprehensive, step-by-step guides detailing the procedures for deploying various resources across different cloud service providers. These guides are methodically divided into several stages, encompassing general instructions, installation processes, the creation of deployment templates, the actual deployment of resources to the cloud, subsequent modifications to these resources, and ultimately, their termination. Conversely, the introductory guide for Ansible adopts a different approach, focusing primarily on explicating key concepts such as Playbooks, Plays, Inventories, among others. Notably, Ansible's documentation does not include a direct, hands-on tutorial. However, it compensates for this

by providing several GitHub links that direct users to demonstrative examples.

During the actual implementation of the deployment, the search functionality embedded within each tool's documentation page was instrumental in locating the necessary resources. Utilizing search terms such as "Kubernetes" and "database AWS," we were able to retrieve pertinent documentation from all the tools. However, it is important to note that in the case of Terraform, users are required to conduct searches for different modules on a distinct registry page, as indicated by the URL https://registry.terraform.io/. Additionally, the search preview feature within the Ansible documentation presented some challenges in terms of usability. Specifically, the search results displayed a limited amount of relevant information, as is evident in the figure 5.4. This limitation in the search preview's effectiveness in Ansible documentation necessitated a more thorough examination of the search results to identify the most pertinent resources.

Furthermore, it merits attention that the integration of JSDoc documentation within the Pulumi TypeScript package significantly enhanced our development experience. This integration allowed for the documentation of each module to be directly accessible within our Integrated Development Environment (IDE). This feature considerably reduced the need to alternate between the IDE and the external documentation website, as the pertinent information was readily available within the IDE interface itself. Over an extended period, this aspect of the Pulumi TypeScript package could potentially result in substantial time savings, particularly as it obviates the necessity for users to navigate through external documentation pages to find specific information. This seamless integration of documentation within the development environment represents a noteworthy advancement in facilitating more efficient coding practices.

In conclusion, the analysis of community support and documentation across these Infrastructure as Code (IaC) tools reveals a spectrum of strengths and areas for im-

provement. Terraform, with its extensive documentation, robust Stack Overflow presence, and a significant number of GitHub contributors, leads in this evaluation, earning it the top grade of 1. It exemplifies a strong balance of community engagement and comprehensive documentation. Pulumi follows closely, distinguished by its advanced integration of JSDoc documentation within the TypeScript package and a reasonable level of community interaction, warranting a grade of 2. This grade acknowledges its commendable efforts in documentation accessibility and community support, though it falls slightly behind Terraform in overall community engagement. Ansible, while actively present in community forums and GitHub, faces challenges with its less intuitive search feature in documentation and the absence of direct, hands-on tutorials. These factors place it in the third position with a grade of 3.

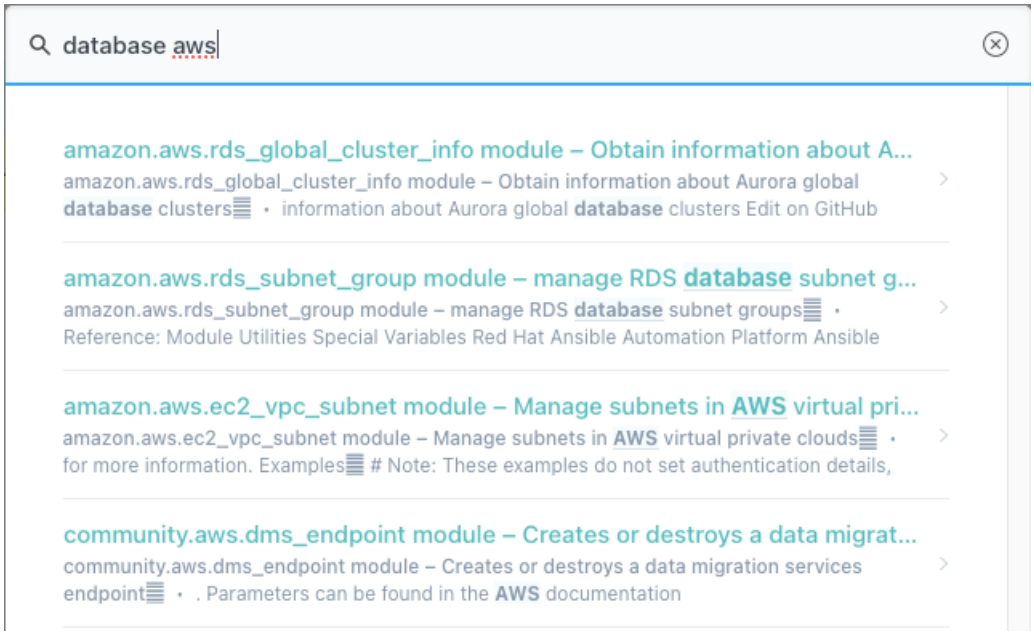

Figure 5.4: Ansible search prompt for "database aws"

Tool	Pulumi	Terraform	Ansible
Learning curve	3	2	1
Scalability and Performance	2	1	3
Security and Compliance	2	1	3
Community Support and Documentation	2	1	1

Table 5.3: Qualitative tool Comparison

5.3.5 Results

Answering the **RQ2**: Enhancing the analysis based on the scoring system for Terraform, Ansible, and Pulumi across various non-technical criteria provides us with a quantitative approach to evaluate their overall performance. With Terraform leading the comparison with a total of 5 points, it highlights its strengths in scalability, performance, security, and community support, though it faces a slightly steeper learning curve compared to Ansible. Ansible secures the second position with 8 points, indicating its accessibility for beginners due to a lower learning curve and strong community support, but it trails in scalability and security features. Pulumi, with a close total of 9 points, underscores its potential in offering powerful flexibility through familiar programming languages, despite the challenges posed by its learning curve and scalability in very large deployments.

The narrow point difference between Ansible and Pulumi suggests a nuanced comparison where the choice between the two may depend significantly on specific project requirements and user expertise. For instance:

- For teams with strong programming skills and a need for customizable infrastructure management, Pulumi's approach may offer more direct benefits, despite its slightly higher overall point score.

- Conversely, Ansible may be preferred for its simplicity and ease of use, partic-

ularly in environments where configuration management and automation are the primary concerns.

This scoring system provides a valuable overview, yet it's crucial to recognize that the effectiveness of an IaC tool cannot be fully captured through a quantitative assessment alone. Detailed considerations, such as the specific nature of the infrastructure, team expertise, and particular project requirements, play a critical role in tool selection. For example:

- Teams with a background in software development might leverage Pulumi's use of programming languages to create more dynamic and complex infrastructure deployments.
- Organizations focusing on rapid deployment and configuration with minimal learning investment might prefer Ansible.

Therefore, while the points provide a general indication of performance across selected criteria, a deeper dive into specific use cases, performance benchmarks, and community feedback is essential for an informed tool selection process. Ultimately, the best choice of tool is one that aligns with the team's skills, project requirements, and operational priorities, emphasizing that the "best performing" tool is context-dependent.

5.4 Integration Comparison

In the preceding sections of this thesis, we delineated nine distinct approaches to integrating with cloud service providers. Our methodology encompassed the utilization of pre-established packages, either developed by the creators of these tools or contributed by the wider community. For instance, in the process of deploying resources to AWS via Terraform, we employed the specialized AWS package from

Terraform's suite. These packages facilitate a seamless interaction with the respective cloud provider's API or other technologies such as Kubernetes. IaC tools, as demonstrated in the research segment of this thesis, can independently serve as instrumental aids in provisioning various resources. Nevertheless, in practical applications, particularly in real-world scenarios, these tools are frequently assimilated into broader automation frameworks, such as CI/CD pipelines. An additional integration in our case study involved the synchronization with the cloud provider's authentication mechanisms. Drawing on the empirical data obtained from our test deployments, we will proceed to scrutinize the integration capabilities of these tools. This analysis will specifically focus on their interoperability with cloud providers' APIs, their adaptability within CI/CD pipeline environments, and their compatibility with cloud providers' authentication protocols. Then we are going to utilize the same method as we did in the qualitative comparison to give these tools points based on that how well they integrate with different technologies.

5.4.1 Provider Integration

The integration of cloud providers in various infrastructure as code (IaC) tools exhibits a certain degree of homogeneity. These tools universally employ packages, modules, or plugins to facilitate communication between the respective tool and cloud services. For instance, Ansible utilizes Python modules, Pulumi employs NPM packages (particularly when JavaScript or TypeScript is used), and Terraform relies on binaries written in Go. These modules are responsible for interfacing with consistent application programming interfaces (APIs), resulting in a uniformity of data exchange when similar types of infrastructure are provisioned. Consequently, this study aims to analyze the scope and diversity of these plugins across each tool to gauge the breadth of modules available for each.

Terraform's ecosystem offers a comprehensive module repository available at the

Terraform Registry [48]. This registry encompasses a wide array of modules, including official, partner, and community-developed options. The community modules are predominantly volunteer-created, and while many are listed in the official Terraform registry, some remain external to it. For the purpose of this analysis, which focuses on provider integration, we will enumerate only the "provider" modules. As of the current assessment, Terraform boasts 34 official provider modules, 325 partner modules, and 3517 community modules. Notably, certain community modules may overlap with official provider and partner modules.

Pulumi's module database is accessible via the Pulumi Registry [49]. The Pulumi packages are categorized into Native Providers, Bridged Providers, and Components. Native Providers are built on Pulumi's unique resource model, Bridged Providers leverage existing resource providers from alternative ecosystems (e.g., Terraform), and Components are specialized packages tailored for specific use cases, analogous to Terraform modules. In the context of provider integration analysis, we shall consider only the Native and Bridged Providers. Currently, Pulumi offers 16 Native Providers and 135 Bridged Providers, most of which are developed and maintained by Pulumi.

Ansible, on the other hand, distributes its cloud provider plugins in the form of collections. Users can access these collections via distribution servers such as Ansible Galaxy or a Pulp 3 Galaxy server. Collections can be explored on the Ansible Galaxy website [50]. However, due to the absence of a use case-based filtering option on Ansible Galaxy, this analysis will reference the collections enumerated in the official Ansible community documentation [51]. These collections are either directly managed by Ansible or by community contributors. According to the documentation, there are 103 distinct collections available.

5.4.2 Authentication Tool Integration

Terraform, when deployed in a local environment, leverages provider modules corresponding to GCP, Azure, and AWS. With GCP and Azure, these modules facilitate direct authentication with the respective cloud provider's Command Line Interface (CLI) tools [52] [53]. In the context of AWS, Terraform necessitates the explicit declaration of authentication parameters within the provider block, environment variables, or via an authentication file [54]. Conversely, for GCP and Azure, Terraform supports all the aforementioned method. All three provider modules (GCP, Azure, and AWS) are designed to seamlessly integrate with the cloud provider's authentication mechanisms when executed within the cloud provider's native environment. For example, executing Terraform within an Azure Kubernetes Service (AKS) cluster allows the tool to leverage AKS Workload Identity for authentication purposes.

Pulumi mirrors the authentication approaches of Terraform for GCP, Azure, and AWS. This similarity extends to both the native and bridged provider modules in Pulumi, likely attributable to the fact that many of Pulumi's cloud provider modules are adaptations or 'bridges' of their Terraform counterparts. [55] [56] [57]

In contrast, Ansible diverges in its approach to cloud provider authentication. Unlike Terraform and Pulumi, Ansible does not facilitate direct authentication with the cloud providers' CLI tools. Instead, for all three cloud platforms (GCP, Azure, and AWS), Ansible requires that authentication parameters be supplied either as arguments within Ansible itself or set as environment variables. [58] [59] [60]

5.4.3 CI/CD Integration

In the development of CI/CD pipelines, it is customary to employ shell scripts for orchestrating various stages of the pipeline. The methodology for constructing these pipelines is significantly influenced by the choice of CI/CD tool. A prevalent

approach involves the delineation of pipeline processes in either YAML or JSON formats. As elucidated in the preceding section titled "Authentication Tool Integration," IaC tools discussed in this thesis exhibit compatibility with the transmission of variables either as environment variables or as command-line arguments. This capability is deemed to be sufficiently versatile for the majority of operational scenarios, thereby implying minimal variation in the manner in which these tools are integrated into CI/CD pipelines.

5.4.4 Results

In the course of our examination of these tools, we identified significant distinctions in their approaches to integrating with cloud providers' APIs and their mechanisms for handling authentication with each provider. Each tool has developed its unique method of interfacing with cloud services, reflecting in their respective strengths and limitations in real-world application scenarios. Regarding CI/CD pipeline integration, our analysis revealed that despite the differences in API and authentication integration, all three tools exhibit a high level of adaptability to CI/CD environments.

In answering the **RQ3**, based on the findings in this thesis, it is evident that Terraform, Pulumi, and Ansible each bring unique approaches to the table regarding their integration with other tools and systems for building reusable cloud infrastructure. Terraform stands out for its vast provider ecosystem and robust module support. Pulumi offers integration capabilities through its bridging of Terraform providers and native support, catering well to environments where existing Terraform resources are leveraged alongside Pulumi's unique features. Ansible, while not as deeply integrated into cloud provisioning as Terraform or Pulumi, excels in automation and orchestration across cloud services, making it a valuable tool in scenarios that extend beyond infrastructure provisioning.

6 Discussion

The distinction among Infrastructure as Code (IaC) tools is notable. This research underscores the varying abstraction levels at which these tools operate. Ansible presents a more granular approach, affording users considerable control over task sequencing, variable management, and state handling. Conversely, Pulumi adopts an automated framework, directing users towards specific methodologies. Terraform, interestingly, positions itself between these extremes, balancing user autonomy with guided procedures. This spectrum of control underscores a fundamental principle: increased user privileges necessitate heightened responsibility for ensuring operational efficacy and integrity in cloud infrastructure deployment.

From the vantage point of the researcher, as informed by empirical observations accrued throughout the course of this study, it becomes evident that the intrinsic complexity associated with resource management in cloud environments is not predominantly dictated by the specificities of the tool employed. Rather, the paramount factor influencing this complexity is the depth and breadth of the user's expertise in cloud technologies. The tools in question, while they do offer a facilitative framework for the adept practitioner, are not in themselves a panacea for the challenges encountered. Although this research delineates and contrasts various tools, it refrains from categorically endorsing any single tool as superior. Instead, this study contributes to the discourse by identifying certain characteristics and functionalities of these tools that should be taken into consideration when making a selection.

These identified attributes have been analyzed in terms of their impact on diverse application scenarios, thereby providing a nuanced understanding that can guide decision-making processes in the selection of appropriate tools for specific cloud resource management contexts.

In this thesis, a high-level examination of the technical and qualitative aspects of Infrastructure as Code (IaC) tools has been conducted. For future research, a more granular comparison of specific features across these tools is recommended. Focusing in depth on a singular feature could yield more insightful results than a broad analysis of the entire framework. For instance, a detailed study comparing the methodologies of task ordering across different IaC tools could elucidate the fundamental operational differences and strengths of each tool. Also, the selection of tools in this thesis is relatively narrow compared to the number of available tools, so a broader selection of tools could result into different findings.

7 Conclusion

In this thesis, we conducted a comprehensive evaluation of the performance characteristics of three Infrastructure as Code (IaC) tools: Terraform, Pulumi, and Ansible. This analysis entailed the creation of three distinct deployments utilizing each of these tools, followed by a detailed examination of the outcomes across three primary dimensions: technical performance, qualitative assessment, and integration capabilities.

We had three different research question in this thesis which were:

RQ1: How do the features and capabilities of different IaC tools compare when it comes to building reusable cloud infrastructure?

RQ2: How do ease of use and learning curve, scalability and performance, security and compliance features, and community support and documentation with different IaC tools compare?

RQ3: How does the integration with other tools and systems compare for different IaC tools when building reusable cloud infrastructure?

Answer that we found for the **RQ1** by utilizing the insights derived from our comprehensive analysis, we discerned variances in the technical dimensions of these tools, predominantly in the ambit of functionalities they proffer. Terraform and Pulumi are distinguished by their robust support for a broad spectrum of features, ranging from resource elimination to intricate state management processes, whereas Ansible is characterized by its more streamlined framework, encompassing a more

restricted array of built-in functionalities. Answer for the **RQ2** through a qualitative evaluation, our findings suggest that Terraform exhibits superior performance on a general scale. However, the demarcation between Ansible and Pulumi appears less pronounced, with their efficacy heavily contingent upon the specific operational contexts in which they are employed. Our findings for the **RQ3** suggest that the disparities in integration capabilities among these tools emerged as minimal throughout our investigation. The primary differentiator was identified as the extent of available modules and plugins, with Terraform securing the lead in terms of the volume of modules furnished by both its maintainers and the extensive community. This nuanced comparative study underscores the importance of contextual application in discerning the most suitable tool for given infrastructure management tasks.

In a general level, our findings in the chapter 5 suggest that there is no universally superior tool; rather, the performance of each tool varies significantly across different operating environments. These results underscore the importance of aligning the selection of an IaC tool with the specific requirements and constraints of the intended operating environment, rather than making a decision based solely on the comparative features of the tools. This approach advocates for a more nuanced and environment-specific selection process, which is critical for optimizing infrastructure management and automation in diverse IT ecosystems.

The constraints of this thesis are primarily twofold: the limited range of tools analyzed and the methodology employed for selecting these tools. Our analysis was confined to a relatively small sample, restricting the generalizability of our findings across the broader spectrum of available infrastructure management tools. Furthermore, the selection process for these tools was significantly influenced by their popularity, serving as one of the principal criteria. This approach, while practical for identifying widely-used tools, may overlook emerging or niche tools that could offer unique insights or superior functionalities in specific contexts. Additionally,

relying predominantly on popularity may inadvertently bias the study towards tools that benefit from greater marketing resources or community support, rather than those necessarily offering the best technical capabilities or innovation. Expanding the selection criteria to include a more diverse array of tools and employing a more rigorous methodological framework for selection could potentially enrich the findings and provide a more comprehensive overview of the landscape of infrastructure management tools.

References

[1] K. Morris, *Infrastructure as code: managing servers in the cloud.* " O'Reilly Media, Inc.", 2016.

[2] M. Artac, T. Borovssak, E. Di Nitto, M. Guerriero, and D. A. Tamburri, "Devops: Introducing infrastructure-as-code", in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, IEEE, 2017, pp. 497–498.

[3] M. Guerriero, M. Garriga, D. A. Tamburri, and F. Palomba, "Adoption, support, and challenges of infrastructure-as-code: Insights from industry", in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, IEEE, 2019, pp. 580–589.

[4] L. Qian, Z. Luo, Y. Du, and L. Guo, "Cloud computing: An overview", in *Cloud Computing: First International Conference, CloudCom 2009, Beijing, China, December 1-4, 2009. Proceedings 1*, Springer, 2009, pp. 626–631.

[5] J. Surbiryala and C. Rong, "Cloud computing: History and overview", in *2019 IEEE Cloud Summit*, IEEE, 2019, pp. 1–7.

[6] K. D. Foote, "Ca brief history of cloud computing", Dataversity, 2021.

[7] N. L. Abdulnabi and R. R. Asaad, "Challenges and benefits of cloud computing: Comparison study", *Academic Journal of Nawroz University*, vol. 11, no. 4, pp. 345–350, 2022.

[8] S. Achar and N. L. Tisuela, "A review of hosting enterprise saas with iac on multi-cloud platforms", *International Journal of Reciprocal Symmetry and Physical Sciences*, vol. 7, pp. 14–23, 2020.

[9] A. Sether, "Cloud computing benefits", *Available at SSRN 2781593*, 2016.

[10] B. Russo, L. Valle, G. Bonzagni, D. Locatello, M. Pancaldi, and D. Tosi, "Cloud computing and the new eu general data protection regulation", *IEEE Cloud Computing*, vol. 5, no. 6, pp. 58–68, 2018.

[11] *Infographic: Amazon Maintains Lead in the Cloud Market*, [Online; accessed 9. Dec. 2023], Dec. 2023. [Online]. Available: `https://www.statista.com/chart/18819/worldwide-market-share-of-leading-cloud-infrastructure-service-providers`.

[12] P. Wankhede, M. Talati, and R. Chinchamalatpure, "Comparative study of cloud platforms-microsoft azure, google cloud platform and amazon ec2", *J. Res. Eng. Appl. Sci*, vol. 5, no. 02, pp. 60–64, 2020.

[13] N. Jain and S. Choudhary, "Overview of virtualization in cloud computing", in *2016 Symposium on Colossal Data Analysis and Networking (CDAN)*, IEEE, 2016, pp. 1–4.

[14] A. P. Rajan *et al.*, "Evolution of cloud storage as cloud computing infrastructure service", *arXiv preprint arXiv:1308.1303*, 2013.

[15] V. Bucur, C. Dehelean, and L. Miclea, "Object storage in the cloud and multi-cloud: State of the art and the research challenges", in *2018 ieee international conference on automation, quality and testing, robotics (aqtr)*, IEEE, 2018, pp. 1–6.

[16] I. Arora and A. Gupta, "Cloud databases: A paradigm shift in databases", *International Journal of Computer Science Issues (IJCSI)*, vol. 9, no. 4, p. 77, 2012.

[17] S. K. Mishra, B. Sahoo, and P. P. Parida, "Load balancing in cloud computing: A big picture", *Journal of King Saud University-Computer and Information Sciences*, vol. 32, no. 2, pp. 149–158, 2020.

[18] J. Wack, K. Cutler, and J. Pole, "Guidelines on firewalls and firewall policy", *NIST special publication*, vol. 800, p. 41, 2002.

[19] A. Bhardwaj and C. R. Krishna, "Virtualization in cloud computing: Moving from hypervisor to containerization—a survey", *Arabian Journal for Science and Engineering*, vol. 46, no. 9, pp. 8585–8601, 2021.

[20] J. Schleier-Smith, V. Sreekanti, A. Khandelwal, *et al.*, "What serverless computing is and should become: The next phase of cloud computing", *Communications of the ACM*, vol. 64, no. 5, pp. 76–84, 2021.

[21] I. A. Mohammed, "Cloud identity and access management–a model proposal", *International Journal of Innovations in Engineering Research and Technology*, vol. 6, no. 10, pp. 1–8, 2019.

[22] H. Borges, A. Hora, and M. T. Valente, "Understanding the factors that impact the popularity of github repositories", in *2016 IEEE international conference on software maintenance and evolution (ICSME)*, IEEE, 2016, pp. 334–344.

[23] K. Shirinkin, *Getting Started with Terraform*. Packt Publishing Ltd, 2017.

[24] *Command: graph | Terraform | HashiCorp Developer*, [Online; accessed 9. Dec. 2023], Dec. 2023. [Online]. Available: `https://developer.hashicorp.com/terraform/cli/commands/graph`.

[25] *Command: init | Terraform | HashiCorp Developer*, [Online; accessed 23. Oct. 2023], Oct. 2023. [Online]. Available: `https://developer.hashicorp.com/terraform/cli/commands/init`.

[26] *Command: apply | Terraform | HashiCorp Developer*, [Online; accessed 23. Oct. 2023], Oct. 2023. [Online]. Available: https://developer.hashicorp.com/terraform/cli/commands/apply.

[27] *Command: destroy | Terraform | HashiCorp Developer*, [Online; accessed 23. Oct. 2023], Oct. 2023. [Online]. Available: https://developer.hashicorp.com/terraform/cli/commands/destroy.

[28] *About*, [Online; accessed 9. Dec. 2023], Dec. 2023. [Online]. Available: https://www.pulumi.com/about.

[29] Pulumi. "Pulumi Documentation: Atlassian". Accessed: 3.10.2023. (), [Online]. Available: https://www.pulumi.com/case-studies/atlassian/.

[30] Pulumi. "Pulumi Documentation: Converters". Accessed: 3.10.2023. (), [Online]. Available: https://www.pulumi.com/docs/using-pulumi/adopting-pulumi/converters/.

[31] Pulumi. "Pulumi Documentation: Crossguard". Accessed: 3.10.2023. (), [Online]. Available: https://www.pulumi.com/docs/using-pulumi/crossguard/.

[32] Pulumi. "Pulumi Documentation: AwsGuard". Accessed: 3.10.2023. (), [Online]. Available: https://www.pulumi.com/docs/using-pulumi/crossguard/awsguard/.

[33] *Concepts*, [Online; accessed 14. Feb. 2024], Feb. 2024. [Online]. Available: https://www.pulumi.com/docs/concepts.

[34] Pulumi. "Pulumi Documentation: Packages". Accessed: 3.10.2023. (), [Online]. Available: https://www.pulumi.com/docs/using-pulumi/pulumi-packages/.

[35] Pulumi. "Pulumi Documentation: Secrets". Accessed: 3.10.2023. (), [Online]. Available: https://www.pulumi.com/docs/concepts/secrets/.

[36] *pulumi new*, [Online; accessed 9. Dec. 2023], Dec. 2023. [Online]. Available: https://www.pulumi.com/docs/cli/commands/pulumi_new.

[37] *pulumi up*, [Online; accessed 9. Dec. 2023], Dec. 2023. [Online]. Available: https://www.pulumi.com/docs/cli/commands/pulumi_up.

[38] *pulumi destroy*, [Online; accessed 9. Dec. 2023], Dec. 2023. [Online]. Available: https://www.pulumi.com/docs/cli/commands/pulumi_destroy.

[39] M. Heap, *Ansible: from beginner to pro*. Springer, 2016.

[40] Y. Omofoyewa, A. Grebe, and P. Leusmann, "Iac reusability for hybrid cloud environment",

[41] ardalis, *Common web application architectures - .net*, Mar. 2023. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/common-web-application-architectures.

[42] *masters*, [Online; accessed 7. Apr. 2024], Apr. 2024. [Online]. Available: https://github.com/nuksuli/masters.

[43] *Config*, [Online; accessed 28. Oct. 2023], Oct. 2023. [Online]. Available: https://www.pulumi.com/docs/concepts/config.

[44] *Case Study: Modernizing Manufacturing with 4IR and Pulumi*, [Online; accessed 9. Dec. 2023], Dec. 2023. [Online]. Available: https://www.pulumi.com/case-studies/4ir.

[45] *Panther Labs: Increasing Velocity & Innovation*, [Online; accessed 9. Dec. 2023], Dec. 2023. [Online]. Available: https://www.pulumi.com/case-studies/panther-labs.

[46] W. Collin *et al.*, "Automation in multi-domain software-defined networking: Overview and use cases", 2021.

[47] I. Moutidis and H. T. Williams, "Community evolution on stack overflow", *Plos one*, vol. 16, no. 6, e0253010, 2021.

[48] *Browse Providers | Terraform Registry*, [Online; accessed 5. Mar. 2024], Mar. 2024. [Online]. Available: `https://registry.terraform.io/browse/providers`.

[49] *Pulumi Registry*, [Online; accessed 5. Mar. 2024], Mar. 2024. [Online]. Available: `https://www.pulumi.com/registry`.

[50] *Ansible Galaxy - Collections*, [Online; accessed 5. Mar. 2024], Mar. 2024. [Online]. Available: `https://galaxy.ansible.com/ui/collections`.

[51] *Collection Index — Ansible Documentation*, [Online; accessed 5. Mar. 2024], Feb. 2024. [Online]. Available: `https://docs.ansible.com/ansible/latest/collections/index.html`.

[52] *Docs overview | hashicorp/google | Terraform | Terraform Registry*, [Online; accessed 28. Jan. 2024], Jan. 2024. [Online]. Available: `https://registry.terraform.io/providers/hashicorp/google/latest/docs`.

[53] *Docs overview | hashicorp/azurerm | Terraform | Terraform Registry*, [Online; accessed 28. Jan. 2024], Jan. 2024. [Online]. Available: `https://registry.terraform.io/providers/hashicorp/azurerm/latest/docs`.

[54] *Docs overview | hashicorp/aws | Terraform | Terraform Registry*, [Online; accessed 28. Jan. 2024], Jan. 2024. [Online]. Available: `https://registry.terraform.io/providers/hashicorp/aws/latest/docs`.

[55] *AWS Classic Installation & Configuration*, [Online; accessed 28. Jan. 2024], Jan. 2024. [Online]. Available: `https://www.pulumi.com/registry/packages/aws/installation-configuration`.

[56] *Google Cloud (GCP) Classic Installation & Configuration*, [Online; accessed 28. Jan. 2024], Jan. 2024. [Online]. Available: `https://www.pulumi.com/registry/packages/gcp/installation-configuration`.

[57] *Azure Classic Installation & Configuration*, [Online; accessed 28. Jan. 2024], Jan. 2024. [Online]. Available: https://www.pulumi.com/registry/packages/azure/installation-configuration.

[58] *Amazon Web Services Guide — Ansible Documentation*, [Online; accessed 28. Jan. 2024], Jan. 2024. [Online]. Available: https://docs.ansible.com/ansible/latest/collections/amazon/aws/docsite/guide_aws.html.

[59] *Google Cloud Platform Guide — Ansible Documentation*, [Online; accessed 28. Jan. 2024], Jan. 2024. [Online]. Available: https://docs.ansible.com/ansible/latest/scenario_guides/guide_gce.html.

[60] *Microsoft Azure Guide — Ansible Documentation*, [Online; accessed 28. Jan. 2024], Jan. 2024. [Online]. Available: https://docs.ansible.com/ansible/latest/scenario_guides/guide_azure.html.